MASTER EN SCIENCES INFORMATIQUES

APIzation

proposition d'une méthodologie et établissement d'un modèle d'affaires

Timmermans, Lionel

Award date:
2015

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2014-2015

APIzation : proposition d'une méthodologie et établissement d'un modèle d'affaires

-

Lionel Timmermans

Promoteur : _____(Signature pour approbation du dépôt – REE art. 40)

Philippe Thiran

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

RÉSUMÉ

Le but de ce mémoire est d'apporter un éclairage approfondi sur la procédure désignée par le terme « APIzation », d'investiguer les domaines impliqués et impactés par celle-ci, d'en préciser les buts et de détailler les étapes à suivre pour mettre en place une APIzation. Pour ce faire, nous commencerons par définir plusieurs concepts de base ayant un lien étroit avec ce terme, ce qui nous permettra alors de proposer une définition et une explication précises de l'APIzation. Ensuite, à l'aide du Business Model Canvas d'Alexander Osterwalder, nous aborderons les répercussions que l'APIzation peut introduire sur le business modèle d'une entreprise, ainsi que les différents moyens de rémunération qui peuvent être adoptés. Pour finir, nous proposerons une méthode décrivant les étapes clés nécessaires pour mener à bien un projet d'APIzation du début jusqu'à son terme. Elle nous donnera l'opportunité d'évoquer les défis auxquels il faudra faire face ainsi que les technologies les plus en vogue dans ce domaine.

Mots clés : APIzation, API, Business model, Business Model Canvas, Software as a service, SOA

ABSTRACT

The aim of this master thesis is to provide a thorough insight into the procedure known as "APIzation", to investigate the areas involved and affected by it, to clarify the goals and detail the steps to implement an APIzation. To do this, we first define several basic concepts closely associated with this term, which will then allow us to offer a precise definition and explanation of the APIzation. Then, using the Business Model Canvas Alexander Osterwalder, we will address the repercussions that the APIzation can introduce on the business model of a company, as well as various means of compensation that can be adopted. Finally, we propose a method that describes the key steps required to carry out a project of APIzation the beginning to the end. It will give us the opportunity to discuss the challenges to be faced and the hottest technologies in this field.

Keywords : APIzation, API, Business model, Business Model Canvas, Software as a service, SOA

AVANT-PROPOS

Je tiens à exprimer mes sincères remerciements à toutes les personnes qui ont contribué à l'élaboration de mon mémoire. Je tiens tout particulièrement à remercier M. Philippe Thiran, promoteur de ce mémoire, pour m'avoir donné la possibilité d'effectuer ce travail, pour son aide, ses conseils et sa disponibilité. Je remercie également les professeurs et assistants de m'avoir transmis leurs connaissances et savoir-faire au cours de ces années d'études.

Je remercie également ma famille et mes amis pour tout leur soutien durant mon cursus.

TABLE DES MATIÈRES

GLOSSAIRE

Datacenter

Centre dans lequel se trouve toute l'infrastructure matérielle (serveurs, réseaux, etc.) d'une ou plusieurs entreprises. Ce type de centre est conçu pour avoir un contrôle sur l'environnement (climatisation, incendie, etc.), une alimentation redondante en cas de soucis et une sécurité physique élevée.

Data Warehouse

Le terme data warehouse désigne une base de données utilisée à des fins de collecte, d'ordonnancement, de journalisation et de stockage d'informations en provenance de base de données opérationnelles au sein d'une entreprise et sert de base à l'informatique décisionnelle.

CRM (Customer Relationship Management)

Un CRM est un outil informatique utilisé par les entreprises dans le but d'optimiser la relation avec les clients, fidéliser les clients et augmenter le chiffre d'affaires. Il permet de traiter directement avec les clients, sur plusieurs plans tels que la vente, le marketing et le service.

IaaS (Infrastructure as a Service)

Service Cloud donnant accès à une infrastructure de matériels informatiques (serveurs, réseaux, stockage, etc.).

Métalangage

Un métalangage est un formalisme conçu pour décrire la syntaxe et la sémantique d'un langage.

Méta-protocole

Un méta-protocole est un protocole qui décrit d'autres protocoles.

QoS (Quality of Service)

La qualité de service désigne la capacité à fournir un service dans de bonnes conditions en termes de disponibilité, de temps de réponse, de bande passante, débit, gigue, taux de perte de paquets, etc.

SDLC (Systems/Softwares Development LifeCycle)

Le cycle de vie de développement logiciels/systèmes est un modèle conceptuel utilisé dans la gestion de projet. Il décrit les étapes parcourues dans un projet de développement d'un système/logiciel.

SLA (Service Level Agreement)

Contrat garantissant un certain niveau de qualité d'un service entre le fournisseur et l'utilisateur.

Startup

Une startup est une jeune entreprise innovante à fort potentiel de croissance dans le secteur des nouvelles technologies. Elle fait la plupart du temps l'objet de collecte de fonds.

Vendor lock-in

Un fournisseur décide volontairement de mettre ses clients dans une situation d'enfermement propriétaire afin de limiter la possibilité de changer de fournisseur.

Virtualisation

La virtualisation est un mécanisme qui permet de faire fonctionner plusieurs systèmes d'exploitation ou applications sur un seul ordinateur/système d'exploitation. On parle alors de serveur privé virtuel ou d'environnement virtuel.

1

INTRODUCTION

Depuis une dizaine d'années, de grands éditeurs de logiciels se sont lancés dans un nouveau mode de livraison de programmes informatiques, communément désigné par le terme « logiciel en tant que service » (SaaS). Cela leur a permis d'atteindre un nouveau segment de clientèle et de mieux répondre aux besoins des utilisateurs en termes d'efficacité, de rapidité de déploiement et de diminution des coûts (matériels et consommation d'énergie). De nos jours, le marché comprend un grand nombre de SaaS, aussi les utilisateurs ne sont-ils pas en manque de choix. En raison de ce nombre important d'offres SaaS, chaque éditeur recherche des moyens pour se différencier des autres concurrents afin d'être le plus compétitif. De ce fait, des éditeurs ayant développé des logiciels en tant que service, essaient de tirer parti de ce qu'ils ont déjà conçu pour élargir leur segment de clientèle. Aussi une tendance actuelle de ces éditeurs est de mettre à disposition des utilisateurs, des APIs ajoutant ainsi un nouveau canal pour atteindre de nouveaux clients. Lorsque le but recherché n'est pas de fournir des APIs à leurs clients, les éditeurs les utilisent en interne.

Ce mémoire a pour objectif d'expliciter cette tendance qu'ont les éditeurs de logiciels à effectuer une « APIzation » ainsi que de déterminer les parties de l'organisation qui seront impliquées et impactées. De plus, ce travail propose une méthode à suivre pour mener à bien un projet d'APIzation tout en abordant les différents aspects économiques et techniques.

Pour atteindre ce but, nous commencerons par définir et expliquer ce qu'est le Cloud ainsi que plusieurs autres concepts de base, comme les APIs, les Web Services, SOA, tous ayant un lien étroit avec l'APIzation. Ce qui nous permettra, pour conclure cette deuxième partie, de proposer une définition du terme « APIzation » que nous commenterons en détail.

Ensuite, dans la troisième section, nous aborderons les business models que les éditeurs de logiciels pourront adopter lorsqu'ils voudront concevoir des APIs. Pour cela, nous introduirons l'API economy avec ses acteurs et ses avantages. Nous définirons également ce qu'est un business model, nous citerons et décrirons les différents modèles de revenus APIs existants regroupés en plusieurs catégories, chacune de ces catégories se différenciant par leur type principal. Pour finir, nous analyserons les modifications apportées par une APIzation sur le business model d'une entreprise en recourant au Business Model Canvas d'Alexander Osterwalder et Yves Prigneur dont nous procurerons chacun des neufs blocs. Nous nous limiterons à l'entreprise qui aura effectué une APIZation et qui deviendra donc un fournisseur d'APIs.

Dans la quatrième section de notre travail, nous proposerons une méthode à suivre pour effectuer une APIzation. Après avoir soulevé les défis engendrés par la création d'APIs et la

manière de s'y préparer, nous discuterons des approches existantes et nous adopterons celle qui nous parait la plus adéquate à suivre pour le processus d'APIzation. Ensuite, nous exposerons une démarche pour identifier les fonctionnalités d'un programme informatique qu'une entreprise souhaite externaliser. Nous terminerons par deux étapes importantes : la conception des APIs en elles-mêmes et la livraison des APIs. En ce qui concerne la conception, une méthode sera proposée sur base de plusieurs autres trouvées dans la littérature. Les technologies actuellement utilisées seront également présentées. Nous aborderons l'étape de la livraison des APIs en pointant les différences entre un SaaS et le style architectural SOA. Nous verrons comment le style architectural SOA peut être utilisé pour livrer des APIs aux consommateurs et de quelle manière des alternatives comme les plateformes de management APIs en ligne peuvent aider pour effectuer ce travail.

Nous terminerons ce mémoire par une conclusion qui synthétisera l'apport de ce travail. Elle reprendra les points essentiels de chaque section.

Ce travail a été élaboré sur base d'articles trouvés dans la littérature et sur base de présentations effectuées par des personnes ayant une grande expertise dans le domaine.

2

CADRE THÉORIQUE

Dans cette partie, nous commençons par définir et décrire des concepts de base en lien étroit avec l'APIzation, ces concepts sont nécessaires afin de cerner les éléments importants faisant partie du Cloud. Cependant, nous ferons l'impasse sur des détails techniques de certains de ces concepts qui nous écarteraient trop de l'objectif de ce mémoire. Nous proposons ensuite une définition de l'APIzation et nous expliciterons ce terme.

2.1. Concepts de base

2.1.1. Cloud Computing

« Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. » [Mell et al., 2011]

Le Cloud Computing (ou l'informatique en nuage) fait référence à la fois aux applications délivrées comme des services au travers d'internet (typiquement appelés SaaS) ainsi qu'aux matériels et logiciels systèmes des datacenters qui fournissent ces services [Armbrust et al., 2009]. Les matériels et logiciels des datacenters forment ce qu'on appelle un « Cloud ». Quand un Cloud est rendu accessible au grand public par un moyen de paiement à l'usage (« pay-as-you-go »), il s'agit d'un Cloud public et le service vendu est appelé « Utility Computing ». Le Cloud Computing est donc la somme de SaaS et d'Utility Computing. Comme décrit dans le schéma ci-dessous (figure 1), trois acteurs sont identifiés. Le premier est le Cloud Provider qui fournit le Cloud (datacenter composé de matériels et logiciels systèmes). Le second acteur est le Cloud user (l'utilisateur Cloud) qui, grâce à la relation « Utility Computing », disposera des ressources nécessaires pour déployer et exécuter son logiciel qui sera accessible à distance, remplissant ainsi son rôle de Saas Provider (fourniseur Saas). Le dernier acteur est le SaaS User, l'utilisateur de l'application SaaS.

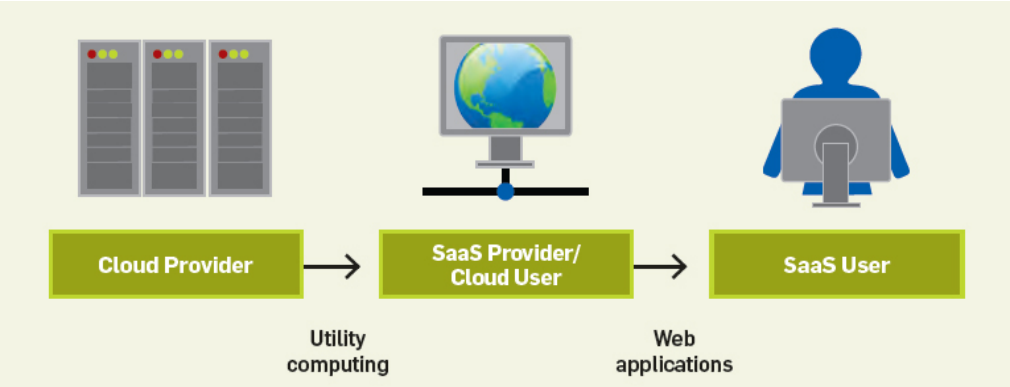

Figure 1 - Cloud Computing : Cloud Provider, SaaS Provider et SaaS User [Armbrust et al., 2009]

Le Cloud Computing comprend cinq caractéristiques essentielles [Mell et al., 2011; Dillon et al., 2010] :

- **Self-service à la demande** : l'utilisateur Cloud est en mesure de gérer lui même, sans l'aide du fournisseur, les services qui lui sont mis à disposition. Il pourra en ajouter ou enlever afin d'obtenir les services nécessaires pour répondre au mieux à ses besoins. Pour cela, les fournisseurs Cloud mettent à disposition des utilisateurs des outils sous forme d'API ou de portail web qui permettent de configurer l'ensemble des services. Par exemple, Microsoft a un portail web pour son Cloud « Azure » où il est possible de configurer des machines virtuelles, des bases de données, des applications web, des machines learning et autre.

- **Accès par le réseau** : les services Cloud sont rendus accessibles par le réseau, celui-ci est habituellement Internet. Des mécanismes standards sont favorisés pour faciliter l'utilisation via des plateformes clients hétérogènes, légères ou lourdes (smartphones, tablettes, ordinateurs portables ou de bureaux).

- **Mise en commun des ressources** : les ressources informatiques (virtuelles et physiques) du fournisseur Cloud sont partagées de manière dynamique entre plusieurs utilisateurs. L'utilisateur/consommateur Cloud n'a aucun contrôle là-dessus et n'en a généralement pas connaissance puisque dans ce cas de figure, les ressources physiques deviennent invisibles pour tous les utilisateurs. Les ressources qui peuvent être partagées entre plusieurs utilisateurs sont : le stockage, la capacité de traitement, la mémoire et la bande passante réseau. La motivation pour mettre en place un tel mécanisme tient à deux facteurs importants : les économies d'échelle et la spécialisation. Un exemple concret est qu'un utilisateur Cloud n'est pas en mesure de dire où (sur quelle unité de stockage physique) seront stockées ses données dans le Cloud.

- **Élasticité rapide** : cette caractéristique du Cloud Computing permet à l'utilisateur Cloud d'augmenter ou diminuer les ressources qui lui sont attribuées. Ce mécanisme peut s'effectuer de manière automatique sur base de la demande de ressources en

temps réel ou bien de manière manuelle par l'utilisateur qui adaptera le volume de ressources dont il a besoin. Cette caractéristique permet aux utilisateurs Cloud de payer pour ce qu'ils utilisent vraiment (ils ne sont pas liés à un contrat sur des ressources fixes) ainsi que d'éviter d'être à court de ressources lors de sollicitations importantes. Aux yeux des utilisateurs, les ressources qui leur sont mises à disposition paraissent infinies. Il leur est facile de pallier à des montées en charge.

- **Service de monitoring** : les systèmes de Cloud Computing contrôlent et optimisent automatiquement les ressources grâce à des outils de monitoring. Cette caractéristique permet en outre de rapporter au fournisseur Cloud ainsi qu'à chaque utilisateur Cloud, des statistiques sur l'utilisation des ressources.

Il existe quatre modèles de déploiement pour une infrastructure Cloud Computing [Mell et al., 2011; Dillon et al., 2010] :

- **Cloud privé** : dans ce modèle, l'utilisation de l'infrastructure Cloud est exclusivement réservée à un organisme, pour un ou plusieurs utilisateurs Cloud faisant partie de celui-ci. L'infrastructure appartiendra soit à l'organisme, soit à un tiers, soit à une combinaison des deux. La motivation pour l'utilisation de ce type de modèle est basée sur plusieurs aspects. Premièrement, il permet de maximiser et d'optimiser l'utilisation des ressources déjà existantes au sein de l'organisation. Deuxièmement, il constitue une bonne solution pour répondre aux problèmes de sécurité, tant pour la protection des données que pour la confiance. Troisièmement, il est encore coûteux de transférer des données depuis l'infrastructure locale vers un Cloud privé. Finalement, les universités construisent souvent des Cloud privés pour la recherche et l'enseignement.

- **Cloud communautaire** : cette infrastructure Cloud est utilisée par plusieurs organisations qui peuvent présenter des besoins communs en terme de processus métier, sécurité, règles, conformités… Elle peut être la propriété d'une ou plusieurs organisations, de tiers ou d'une combinaison des deux. Par exemple, le Cloud communautaire CMed permet de faciliter la mise sur le marché des nouveaux médicaments conçus par les laboratoires médicaux.

- **Cloud public** : le Cloud public vise une utilisation par le grand public. C'est le modèle de déploiement de Cloud le plus dominant. L'infrastructure peut appartenir à une entreprise, une organisation, une école ou une combinaison de ces dernières. Des entreprises telles que Google, Microsoft, Amazon possèdent des Clouds publics. Ces entreprises mettent en place leurs propres règles, valeurs, profits, coûts et modèles de charge auxquels les utilisateurs Cloud doivent se soumettre pour utiliser leur service Cloud.

- **Cloud Hybride** : dans le cas d'un Cloud Hybride, l'infrastructure est composée de deux ou plusieurs infrastructures distinctes (Clouds publics, communautaires, privés). Ces infrastructures distinctes sont liées ensemble grâce à des technologies standards ou propriétaires qui vont permettre un échange de données entre elles ainsi qu'une portabilité des applications (par exemple l'équilibrage de charge entre les Clouds). Ce type de modèle de déploiement peut être utile quand une organisation possède des

données de différents niveaux de confidentialité (par exemple, les données ayant un haut niveau de confidentialité seront stockées sur l'infrastructure du Cloud privée). De plus, ce modèle permet à l'organisation qui l'adopte d'optimiser ses ressources pour augmenter ses compétences en déplaçant les fonctions commerciales périphériques sur le Cloud public et en gardant le contrôle sur les activités de base à travers le Cloud privé.

Le Cloud Computing comporte également trois modèles de services (voir figure 2). Ces modèles sont abordés ci-dessous.

2.1.2. SaaS

« The definition of SaaS typically include both business and technical perspectives, the former beging the dominating viewpoint. […] is currently used for software that is provisioned over the internet and used usually with a web browser. The same naming convention is currently used also for other parts of the computing stack, e.g. Infrastructure-as-a-Service (IaaS) and Platforme-as-a-Service (PaaS).» [Mäkilä et al., 2010]

« SaaS » (Software-as-a-Service) est le premier modèle de service. Ce terme désigne un logiciel qui est distribué aux utilisateurs à la manière d'un service. C'est un composant du Cloud Computing. À la différence d'une installation et d'une exécution sur une machine locale, le programme est installé et exécuté sur un environnement d'hébergement mis en place sur une infrastructure Cloud d'un fournisseur. Les utilisateurs SaaS accèdent aux fonctionnalités du logiciel au moyen d'internet via une interface client léger, telle qu'un navigateur internet, ou une interface de programme. La grande différence réside donc dans le mode de livraison du logiciel. Un très bon exemple est le traitement de texte en ligne Google Docs. Les utilisateurs Cloud n'ont pas à gérer ou contrôler l'infrastructure Cloud sous-jacente qui inclut le réseau, le stockage, les serveurs, les systèmes d'exploitation, les systèmes logiciels, etc. C'est le fournisseur Cloud qui prend la responsabilité quant au bon fonctionnement de l'infrastructure et de la bonne coordination de l'ensemble des composants qui la composent.

2.1.3. PaaS

Le deuxième modèle de service est le « PaaS » (Plateform-as-a-Service). Celui-ci désigne la plateforme de développement supportant l'intégralité du cycle de vie du logiciel qu'un fournisseur Cloud met à disposition des utilisateurs Cloud, afin que ceux-ci puissent développer et/ou déployer leurs logiciels acquis ou construits en utilisant des langages de programmation (Java, .Net, Python...), des librairies et services supportés par le fournisseur [Dillon et al., 2010]. De plus, cet environnement met aussi à disposition des développeurs des outils de tests, de monitoring, des systèmes de gestion de base de données, etc. Des exemples de PaaS sont Google App Engine, Microsoft Azure, etc. Les utilisateurs Cloud n'ont pas à gérer ou contrôler l'infrastructure Cloud sous-jacente (IaaS) qui inclut le réseau, le stockage, les serveurs, les systèmes d'exploitation, etc. Tout est géré automatiquement par le fournisseur Cloud. Le fait que les utilisateurs Cloud doivent utiliser les langages de programmation et les outils qui sont propres au fournisseur Cloud peut engendrer un problème d'emprisonnement propriétaire (« vendor lock-in ») [Tsai et al., 2010].

2.1.4. IaaS

Le troisième et dernier modèle de service est « IaaS » (Infrastructur-as-a-Service). Ce terme fait référence à l'ensemble du matériel qui compose l'infrastructure IT du service « Cloud ». Un fournisseur IaaS met à disposition des fournisseurs PaaS le matériel (processeurs, stockages, réseaux, mémoire vive et autres ressources informatiques) sur lequel la plateforme de développement sera déployée. La virtualisation est très souvent utilisée pour permettre d'intégrer ou décomposer des composants physiques en composants logiques. Un exemple d'IaaS est Amazon EC2.

En résumé, l'IaaS est l'ensemble du matériel informatique des datacenters (processeurs, unités de stockage, réseaux, etc.) qui est mis à disposition tel un service aux utilisateurs. Le PaaS est la plateforme de développement qui sera installée sur les supports physiques IaaS et qui sera offerte comme un service afin d'y installer une ou plusieurs applications. Enfin, le SaaS est l'application qui sera installée et exécutée sur la plateforme de développement PaaS que les utilisateurs finaux utiliseront à distance.

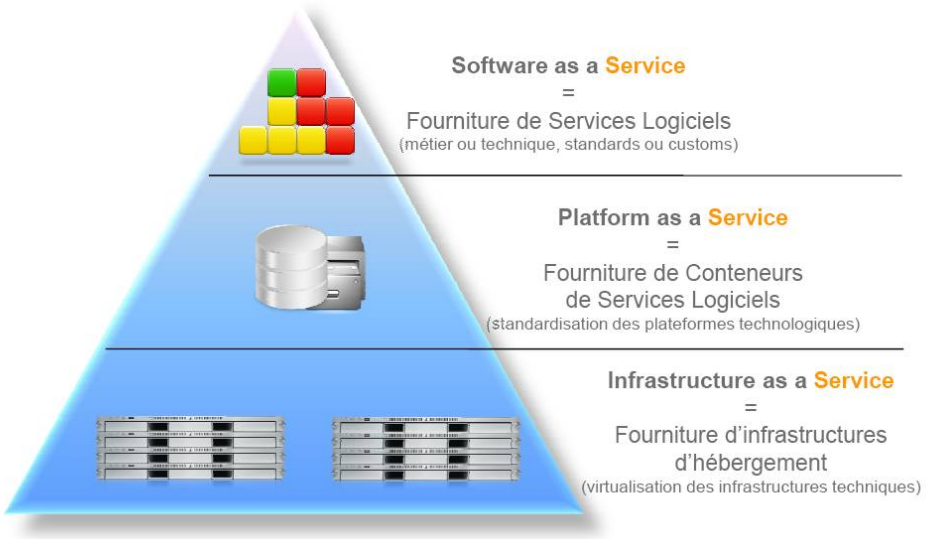

Figure 2 – Pyramide des modèles de services : SaaS PaaS IaaS [Hourdeau et al., 2015]

2.1.5. SOA

2.1.5.1. Définition

« A Service-Oriented Architecture (SOA) is a software architecture that is based on the key concepts of an application frontend, service, service repository, and services bus. A service consists of a contract, one or more interface, and an implementation. » [Krafzig et al., 2004]

SOA est un style architectural pour les logiciels. Il permet à des fournisseurs de services de proposer leurs services à des consommateurs de services via un intermédiaire qui joue le rôle d'un « annuaire ». Par « annuaire », nous désignons le composant qui contient toutes les références aux services qui sont offerts, il permet d'effectuer une recherche d'un service sur

base d'un ou plusieurs critères. Le but de ce style architectural est de garantir l'interopérabilité entre les différents composants (le service lui-même et le programme du consommateur du service), on réduit ainsi le couplage entre ces composants. Les services sont des modules logiciels accessibles par leur nom via une interface, ils sont conçus par des développeurs. Les consommateurs de services sont des logiciels qui intègrent un proxy d'interface de service (c'est une interface qui va jouer le rôle d'interprète entre deux programmes afin qu'ils puissent communiquer) et peuvent ainsi comprendre et utiliser les services sans aucune connaissance détaillée de leurs implémentations. Les principaux piliers de cette architecture sont : la réutilisabilité des services, chaque service doit avoir un contrat, les services doivent être facilement découverts, les services peuvent être composés à partir d'autres services.

Dans le schéma ci-dessous (figure 3) sont représentés les différents composants (artéfacts) de ce style architectural, Application front-end et service étant les composants principaux de l'architecture SOA.

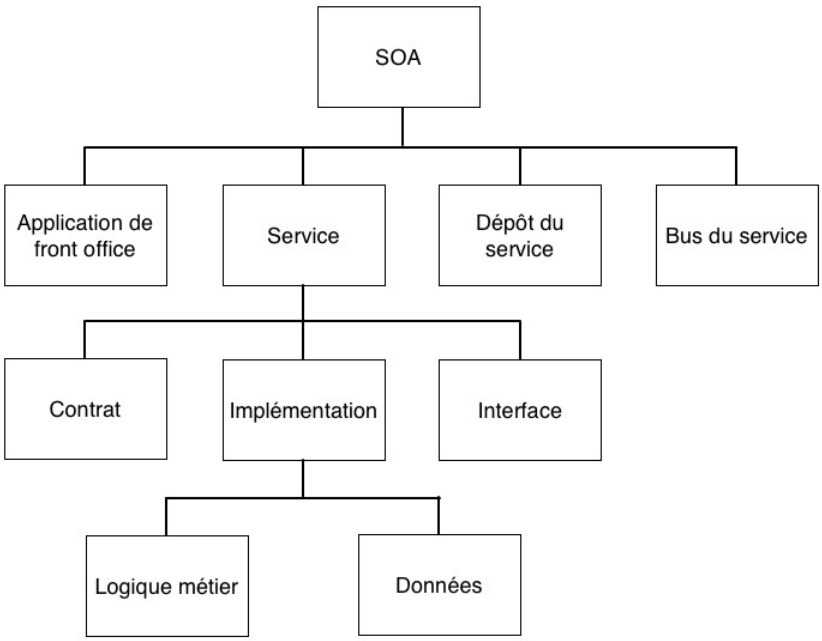

Figure 3 – Artéfacts du style architectural SOA [Krafzig et al., 2004]

2.1.5.2. Caractéristiques

On attend trois caractéristiques d'un service. [Curbera et al., 2001]

- Le service doit être complètement décrit dans son fichier de description. Celui-ci comprend les informations comportementales de l'application et ses caractéristiques non fonctionnelles (telles que la sécurité, la qualité du service, la performance, la disponibilité, etc.)

- Il doit fournir un couplage faible par l'intermédiaire des protocoles d'interopérabilités.

- Il doit offrir une intégration flexible. Une application distante doit pouvoir, via la description du web service, détecter si un protocole de communication plus performant peut être supporté entre deux applications et ainsi faciliter l'extension du niveau d'intégration entre elles.

Pour la description du service et la standardisation, il utilise des messages et des formats de document au lieu d'API. Ceci permet de s'assurer de l'interopérabilité entre les services.

Ci-dessous, la figure 4 illustre les différents acteurs (fournisseur de services, consommateur de services et service de « courtage », nommé « annuaire » plus haut dans le texte) d'une architecture SOA et leurs interactions.

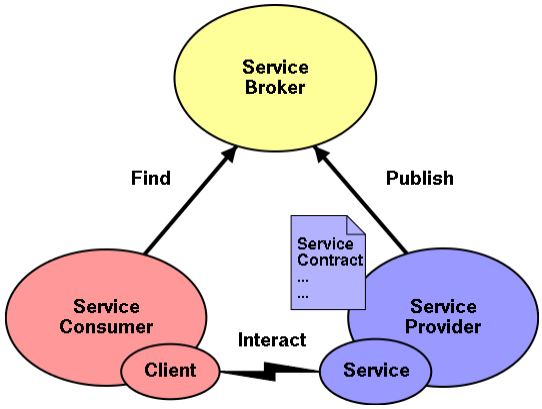

Figure 4 – Interactions entre les composants d'une architecture SOA [Hass, 2015]

Il existe deux types de services. Le premier, appelé « service composite », désigne des services dont l'appel à d'autres services s'avère nécessaire pour fournir la fonctionnalité attendue. Par exemple, un service de réservation en ligne de chambre d'hôtel fera appel à plusieurs autres services pour connaître les hôtels et les chambres disponibles. Le deuxième type est le service dit « service basique ». Il s'agit de services faisant uniquement appel aux composants du système d'information local. Quel qu'il soit, le type de service est totalement invisible du point de vue de l'utilisateur. Aucun changement n'a lieu dans l'utilisation du service, qu'il soit composite ou basique.

Comme énoncé plus haut, le but du style architectural SOA est de garantir l'interopérabilité entre les différents composants (le service lui-même et le programme du consommateur du service). Arrêtons-nous quelques instant à ce concept d'interopérabilité.

2.1.6. Interopérabilité

« Broadly speaking, interoperability can be defined as a measure of the degree to which diverse systems, organizations, and/or individuals are able to work together to achieve a common goal. » [Ide et al., 2010]

Deux types d'interopérabilité dans l'informatique peuvent être distingués [Ide et al. , 2010]:

- l'interopérabilité syntaxique qui a pour but d'assurer une communication et un échange de données en définissant la nature, le type et le format des messages ou données. Elle se base sur les formats de données et les protocoles de communication. Ainsi les systèmes peuvent traiter les données échangées, par contre il n'y a toujours pas de garantie quant à une interprétation identique de ces informations.

- l'interopérabilité sémantique qui assure que la signification exacte des informations échangées soit compréhensible par les systèmes lors de l'interprétation automatique des données. Elle mène à produire des résultats utiles.

Ainsi la définition des informations échangées entre plusieurs systèmes est faite sans ambigüité : « ce qui est envoyé est le même que ce qui est compris » [Ide et al., 2010].

2.1.7. Web Service

2.1.7.1. Définition

« A Web service is a networked application that is able to interact using standard application-to-application Web protocols over well defined interfaces, and which is described using a standard functional description language. » [Curbera et al., 2001]

Un « Web Service » est basé sur les spécifications technologiques des applications web, tandis que le style architectural SOA renvoie à un principe de conception de logiciels.

2.1.7.2. Structure

On pourrait structurer un Web Service en trois grandes parties : description, découverte et interactions [Alonso et al., 2004]. La Figure 5 schématise les deux premières parties (description du service et pile de découverte).

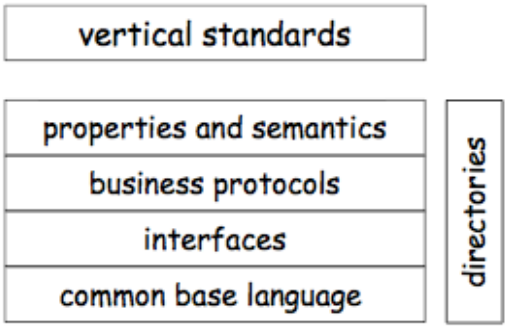

Figure 5 – Description du service et pile de découverte [Alonso et al., 2004]

La « description du service » lui-même comprend cinq niveaux :

- « Le langage de base commun » est le métalangage qui sera adopté comme base pour les langages qui serviront à décrire le service. Actuellement, l'XML est le plus couramment utilisé en raison de sa syntaxe qui permet une grande flexibilité dans la description des services.

- « Les interfaces ». Il est nécessaire d'utiliser un IDL (Interface Description Language) afin de décrire le contexte du web service (ex : l'adresse du service, le protocole de transport, le type de format de données, les opérations possibles, …). L'IDL le plus largement adopté est WSDL [Christensen et al., 2001].

- « Les protocoles business ». Ce niveau décrit les règles qui permettent de définir des « conversations » valides. Une « conversation » est une suite ordonnée d'opérations effectuées par le client à destination du Web Service. Les langages les plus connus pour décrire ces règles sont le WSCL [Banerji et al., 2002] et le WSBPEL [Alves et al., 2007].

- « Propriétés et sémantiques ». Ce niveau fournit une couche additionnelle au Web Service permettant de donner aux consommateurs du service, des informations complémentaires (telles que le coût, la qualité d'un service, une description textuelle du service …). Pour cela on pourra choisir entre la spécification UDDI [Bellwood et al., 2004], WSO2 ou ebXML.

- Les « Standards verticaux » constituent un ensemble de normes. Les couches précédentes étant génériques, on a besoin de spécifier des normes pour définir des interfaces spécifiques, des protocoles, des propriétés et sémantiques que les services offerts doivent supporter. Ce qui permet de faciliter l'utilisation d'outils standards pour effectuer les échanges et ainsi permettre le développement d'applications client qui pourront interagir de manière efficace avec un Web Service conforme à ces normes.

La deuxième partie correspond à la « découverte du service ». Une fois que le service est correctement décrit, on souhaite le rendre disponible aux utilisateurs. Pour ce faire, on utilise des « directories » où seront stockées les descriptions du service. Les utilisateurs pourront alors accéder aux directories et ainsi rechercher et localiser les services dont ils ont besoin. La publication et la découverte d'un service se fait via des APIs qui sont définies par des spécifications UDDI ou ebXML.

La troisième partie comprend « les interactions du service ». Une fois que la liaison entre la description et la découverte du service est effectuée, il est nécessaire de définir un ensemble d'outils et de niveaux d'abstractions pour permettre des interactions entre les Web Services. Ces différents niveaux d'abstractions consistent en un ensemble de normes qui répondent à différents aspects des interactions. Différents types de protocoles composent les interactions du service (figure 6) :

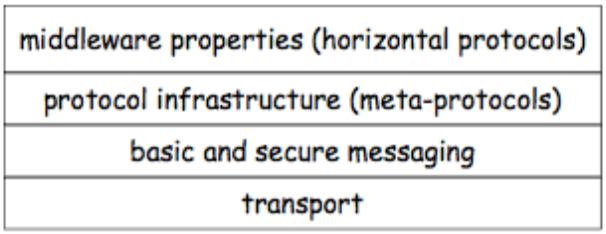

Figure 6 – Pile des interactions du service [Alonso et al., 2004]

- « Transport »: le protocole cache le réseau de communication. Le protocole le plus utilisé pour les Web Service est le HTTP.

- « Message »: un protocole est utilisé pour formater et packager les informations à échanger. À cet effet, SOAP [Gudgin et al., 2007] est le plus largement exploité. Il est possible de renforcer la sécurité au niveau de l'échange de données en utilisant WS-Security [Nadalin et al., 2004].

- « Protocoles d'infrastructure »: un ensemble de méta-protocoles est choisi et exécuté afin de faciliter et coordonner l'exécution des protocoles business. Par exemple, avant qu'une interaction ne débute entre l'utilisateur et le service, ceux-ci ont besoin de se mettre d'accord sur le protocole à utiliser, de déterminer si c'est l'utilisateur ou le service qui coordonne l'exécution du protocole et comment les identificateurs du protocole d'exécution seront inclus dans les messages échangés. On définit ainsi un contexte pour la coordination du service. Pour standardiser ces méta-protocoles et la manière dont WSDL et SOAP doivent être utilisés, on utilise la spécification WS-Coordination [Feingold et al., 2009].

- « Protocoles middleware »: comme les middlewares conventionnels, ceux utilisés pour les Web Services fournissent les même propriétés (ex : fiabilité et transactions). Le middleware est composé de protocoles « horizontaux » du fait qu'ils sont généralement applicables à la plupart des Web Services. Ces protocoles ne sont pas utilisés pour décrire le service, mais pour fournir un ensemble de propriétés de niveau supérieur à toutes sortes d'interactions. Ils peuvent être entièrement gérés par l'infrastructure (c'est la raison pour laquelle ils ne font pas partie de la pile description du service, mais de la pile des interactions du service). Un des premiers protocoles de ce type est WS-Transaction [Cabrera et al., 2002]. Celui-ci permet de définir deux types de coordination [Curbera et al., 2003] : transaction atomique (de courte durée) et transaction business (de longue durée).

2.1.8. API

« An application program interface (API) is a set of routines, protocols, and tools for building software applications. Most operating environments provide an API so that programmers can write applications that are consistent with an operating environment. The APIs typically have documentation that provides instructions for using the API. » [Minnaert, et al, 2002]

Une API constitue une « porte » ouverte sur une fonctionnalité d'une application ou d'un outil, tout en cachant les détails d'implémentation. Elle rend cette fonctionnalité accessible à d'autres développeurs afin qu'ils puissent l'intégrer dans leurs propres logiciels.

Concrètement, une API est composée de classes, de méthodes ou de fonctions, et de types de données. Elle est offerte via une librairie logicielle ou via un service web. Elle est habituellement fournie avec une description qui spécifie comment elle doit être utilisée.
Chaque entreprise conceptrice de programmes informatiques possède un SDLC. Il est donc important d'aborder un point important lié aux APIs : leur cycle de vie. Ce cycle de vie est composé de 4 phases : Plan/Design (Conception), Build/Integrate (Construction), Start use/Manage (Utiliser/Gérer) et Share/Improvement (Distribution/Amélioration). Nous aborderons plus en détail cette partie dans une autre section.

En fonction des besoins de l'éditeur de logiciel, il est possible de distinguer les APIs selon leur accessibilité (3 types) et selon l'écosystème dans lequel les APIs s'intègrent (2 modèles principaux, l'un regroupant deux sous-modèles)

2.1.8.1. Les types d'APIs en fonction de leur accessibilité

La figure 7 illustre les différents niveaux d'accessibilité.

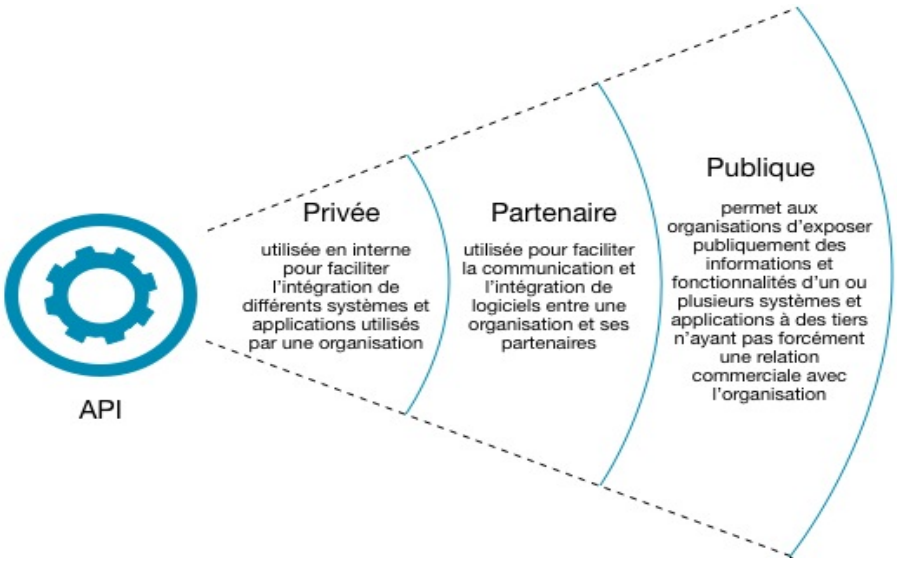

Figure 7 – Classification des niveaux d'accessibilité API

Avec Alagarasan (2015), il est possible de distinguer 3 types d'API :

- Les APIs privées (internes) sont développées par des organisations et ne sont visibles et utilisables qu'uniquement par les développeurs de ces organisations ou ceux qui ont passé un accord avec celles-ci. Elles permettent aux logiciels qui seront créés de tirer parti des systèmes déjà existants dans l'entreprise. En général, les objectifs de ce type d'APIs sont :

diminuer les coûts, rationaliser l'infrastructure, augmenter la flexibilité, améliorer les opérations internes.

- Les APIs publiques sont accessibles aussi bien par les développeurs travaillant pour l'entreprise qui les a conçues que par toutes personnes externes à l'entreprise souhaitant s'enregistrer pour y avoir accès. Le succès d'une API publique est lié en grande partie à la communauté de développeurs qui l'utilisent et qui contribuent à son amélioration. Le but recherché par ce type d'API est la monétisation des actifs exposés de l'entreprise et la stimulation de l'innovation.

- Les APIs partenaires seront visibles et accessibles seulement pour les partenaires de l'entreprise. Dans ce cas-ci, le but recherché est la collaboration de processus ou opérations business. Il arrive cependant qu'une API puisse être à la fois privée et accessible par les partenaires.

2.1.8.2. Les modèles d'APIs selon l'écosystème dans lequel ils s'intègrent

Sur base de l'écosystème dans lequel les éditeurs de logiciels intègrent leurs APIs, nous pouvons identifier deux modèles[1] principaux d'API, l'un regroupant deux sous-modèles.

Le premier modèle identifié est le « modèle fermé » (figure 8).

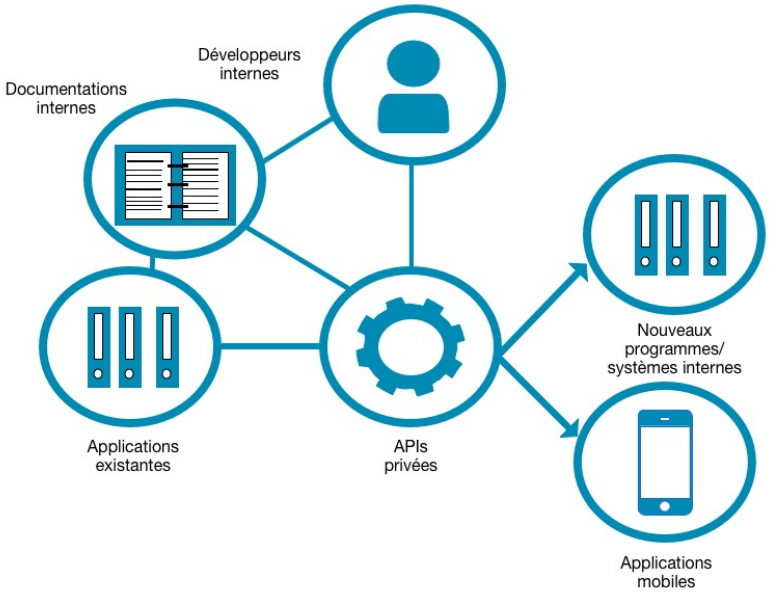

Figure 8 – Modèle fermé. [inspiré et adapté de See Technologies, 2015]

[1] À première vue, les modèles et les types d'APIs peuvent sembler désigner la même chose. Cependant nous introduisons ici le terme « modèle » (on pourrait dire également « écosystème »), car un modèle comprend un type d'API spécifique mais également tous les éléments qui devront être considérés/mis en place avant et lors du processus d'APIzation (business model, gestion de la plateforme API, documentation, API Economy, API management, architecture, etc.). Ces différents aspects seront abordés plus en détail dans les prochaines sections de ce travail.

Ce modèle restreint l'utilisation des APIs exclusivement au sein de l'entreprise, il se repose donc uniquement sur l'utilisation d'APIs privées. Une des raisons pour laquelle un éditeur de logiciels ayant plusieurs applications adopte ce modèle est qu'il souhaite qu'une fonctionnalité particulière, implémentée dans l'une de ses applications, puisse être utilisée par une ou plusieurs autres applications qu'il développe (application de bureau, application mobile, application web, etc.). Afin d'économiser le temps (et donc minimiser le coût) que les développeurs passeront à implémenter cette fonctionnalité dans le programme, il est plus adéquat de réutiliser celle-ci. Dans ce cas-ci, la réutilisabilité constitue donc l'objectif principal.

Le deuxième modèle que nous avons identifié est le « modèle ouvert » (figure 9).

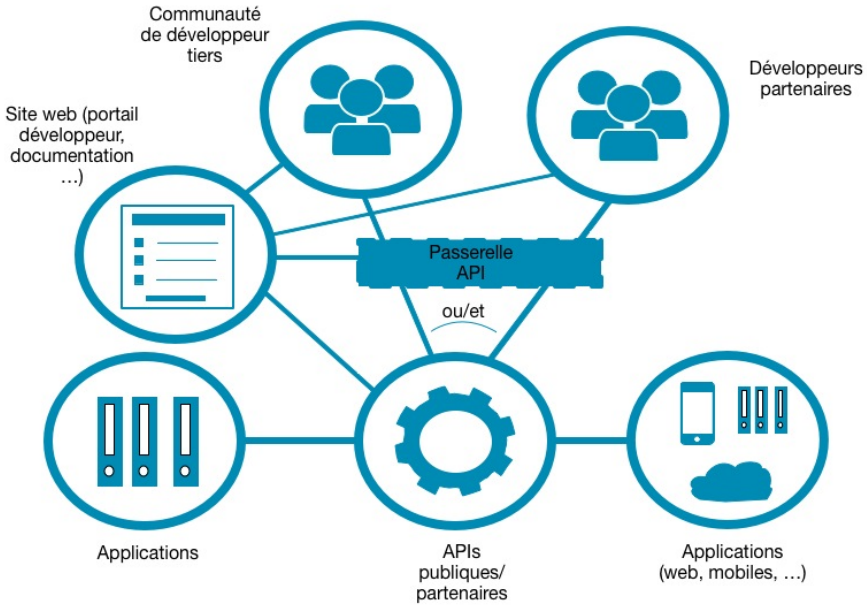

Figure 9 – Modèle ouvert. [inspiré et adapté de See Technologies, 2015]

Ce modèle ouvert vise l'utilisation d'APIs depuis l'extérieur de la société conceptrice de ces APIs. Il se repose sur l'utilisation d'APIs publiques (ou « Open » API) ou partenaires. Ce modèle comprend deux sous-modèles ouverts.

Le premier est le « sous-modèle ouvert universel ». Dans ce sous-modèle, on utilise uniquement des APIs publiques. Ce modèle est dit « ouvert universel » car les APIs sont universelles, elles sont conçues et sont utilisables de la même façon pour tous les utilisateurs. Ce type de sous modèle comprend les caractéristiques de l' « Open » à savoir : la disponibilité et l'accessibilité, la réutilisation et la distribution, la participation universelle. Il est courant de penser à la gratuité lorsque l'on utilise le mot « Open » cependant ce n'est pas du tout le cas, un logiciel peut être open tout en étant payant [Open Source Initiative, 2015]. Ce sous-modèle comprend également l'élaboration d'un business model pour l'éditeur de logiciels, l'adoption de l'API Economy, la mise en place d'une stratégie marketing, etc. Ces éléments seront discutés plus loin.

Plusieurs raisons peuvent amener un éditeur de logiciels à adopter ce sous-modèle ouvert universel. Une première raison est l'intégration d'une application SaaS que l'éditeur a conçue au sein de l'infrastructure IT de l'utilisateur. En effet, cette intégration est connue pour être, dans certains cas, assez laborieuse. De nombreux articles dans la littérature abordent ce problème [Kim et al., 2012; Chou et al., 2010]. L'entreprise doit la plupart du temps pouvoir interconnecter ses différents systèmes d'informations existants (data warehouses, système de facturation, système de sauvegarde, applications sur site …) avec l'application SaaS. Dans de nombreux cas, l'intégration se fera donc sur plusieurs couches et d'une manière que l'éditeur de l'application n'avait pas initialement prévue. En raison de ces différentes complications, cette étape, pourtant essentielle pour le succès de l'application, devient vite pénible et engendre de nombreux problèmes. On parle d'ailleurs d'un coût d'intégration pouvant atteindre 30 à 45% du coût global de l'implémentation SaaS. Cela a pour conséquence d'augmenter la durée du projet [Hai et al., 2009]. Afin de pallier à ce problème, le fournisseur de l'application SaaS pourrait envisager de mettre à la disposition de ses clients des APIs donnant accès aux fonctionnalités de l'application. L'intégration des fonctionnalités dans l'infrastructure IT du client sera alors fortement facilitée grâce aux APIs. Ces APIs seront conçues et utilisables de la même manière, quel que soit l'utilisateur.

Une deuxième raison est qu'un éditeur de logiciels ayant déjà une application SaaS souhaite rendre disponibles les fonctionnalités de celle-ci via un nouveau canal dans le but d'élargir sa base de clients. Dans ce cas, l'éditeur continuera de fournir un accès à l'application SaaS aux clients souhaitant utiliser l'ensemble (ou une grande partie) des fonctionnalités qu'elle contient. En plus de cela, il pourra fournir des APIs permettant d'accéder à certaines fonctionnalités bien ciblées aux clients désireux de n'utiliser qu'une partie des fonctionnalités offertes par l'application.

Le deuxième sous-modèle ouvert est le « sous-modèle ouvert restrictif ». Ce sous-modèle se base sur des APIs partenaires. Ces APIs sont conçues pour être utilisées par des partenaires du concepteur d'APIs. Dans ce cas-ci, elles ne sont pas universelles, chacune d'entre elles est adaptée aux besoins métier de chaque partenaire. Ce sous-modèle comporte également un business model spécifique.

Une raison justifiant qu'un éditeur de logiciels adopte ce sous-modèle ouvert, est le besoin d'externaliser des fonctionnalités d'une application afin que celles-ci puissent être utilisées uniquement par des partenaires dans le cadre d'un contrat commercial. Par exemple, Microsoft et Dropbox ont créé un partenariat afin de pouvoir consulter et modifier des fichiers Microsoft Office depuis Dropbox et, inversement, accéder et sauvegarder des documents sur Dropbox depuis la suite Office. Pour que cela soit possible, ces deux entreprises durent créer des APIs uniquement à destination des partenaires.

Bien sûr, il est tout à fait possible qu'une API publique soit utilisée par des développeurs internes en plus des développeurs externes à l'entreprise conceptrice de l'API. Dans ce cas particulier on se retrouve dans un modèle ouvert universel.

Il existe donc deux modèles principaux : Ouvert et Fermé. Le modèle ouvert comprend deux sous-modèles : l'un restrictif et l'autre universel (figure 10). Chacun de ces (sous)-modèles comprend un type spécifique d'API (publique, privée, partenaire) (tableau 1.).

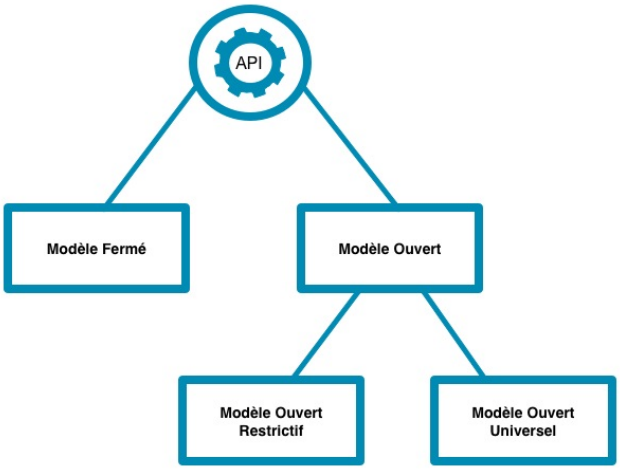

Figure 10 – Les différents modèles

Lorsqu'on se trouve dans un modèle ouvert (restrictif ou universel), il convient de prendre en compte toute une série d'éléments importants tels que la protection, la documentation, un business modèle, les outils de monitoring, l'infrastructure, le service de support. Ces éléments seront abordés plus loin.

Le tableau 1 synthétise les relations entre les types et les modèles d'APIs

		MODÈLE D'APIs		
		Fermé	Ouvert	
			Restrictif	Universel
TYPE D'APIs	APIs privées	X	-	-
	APIs partenaires	-	X	-
	APIs publiques	-	-	X

Tableau 1 – Relations entre les types et les modèles d'APIs

2.2. APIzation

« APization » est un terme utilisé pour désigner le fait d'externaliser des fonctionnalités d'une application existante en créant un ensemble d'APIs. Par « externaliser », nous entendons le fait d'introduire une interface au-dessus d'une fonctionnalité. Cette interface permettra aux utilisateurs d'accéder à une fonctionnalité bien spécifique sans devoir utiliser l'application (et ses fonctionnalités dans son ensemble) comme cela a été conçu initialement. Il peut exister plusieurs niveaux dans l'APIzation. Tout d'abord, une fonctionnalité peut être externalisée de son application pour la rendre utilisable par d'autres programmes au sein de l'infrastructure du concepteur de l'API (typiquement dans un système d'information d'un éditeur de logiciels pour son propre développement), on pourra dans ce cas parler de modèle fermé, ce modèle

fermé se base sur la conception d'APIs privées. À un autre niveau, la fonctionnalité externalisée peut être rendue accessible à des programmes en dehors du concepteur (à des partenaires ou des tierces parties), on parlera alors de modèle ouvert, ce modèle ouvert se base sur la conception d'APIs partenaires et/ou publiques.

Étant donné que le but premier de l'APIzation est la création d'APIs afin de répondre à un besoin particulier d'un éditeur de logiciels, il en découle une mise en place d'une stratégie visant à exploiter correctement les possibilités offertes par ces types et modèles d'APIs.

Après avoir défini son/ses besoin(s), il est nécessaire que l'entreprise adopte l' « API Economy » avant de s'engager pleinement dans la phase de conception des APIs. Dans la prochaine section, nous expliquerons en quoi consiste l'API Economy.

3

ADOPTION D'UNE ÉCONOMIE API ET D'UN BUSINESS MODÈLE

Dans la section précédente nous avons défini et expliqué le terme APIzation ainsi que plusieurs concepts étroitement liés. Nous avons vu que l'APIzation est en lien direct avec l'utilisation d'APIs. Dans cette section nous aborderons les aspects économiques et métiers d'une entreprise lorsque celle-ci se lance dans la création et/ou l'utilisation d'APIs. Nous commencerons par définir et expliquer en quoi consiste le terme « API Economy » et quels sont les acteurs qui y prennent part et les avantages que ceux-ci peuvent en retirer. Ensuite, nous aborderons les modèles de revenus et les classes APIs. Pour terminer cette section, nous discuterons du business model canvas d'une entreprise effectuant une APIzation. Tout au long de cette section nous relèverons les différences entre les APIzations selon les différents modèles d'APIs, et tout particulièrement entre les sous-modèles ouverts.

3.1. API Economy

Trois grands facteurs ont contribué à l'apparition de l'API Economy : la forte croissance du nombre de dispositifs pouvant se connecter à internet (smartphones, tablettes, smartwatch, etc.), les réseaux sociaux et les phénomènes connexes, l'apparition de nouveaux types de logiciels et du marché de programmes informatiques sous forme d'applications (AppStore, GooglePlay, WindowsPhone Store, etc.). Ces différents facteurs permettent l'utilisation des données qu'une entreprise possédait, mais qu'elle n'avait jamais exposées sur une grande échelle. Les APIs sont utilisées pour interconnecter tous ces composants d'une manière simple et efficace.

3.1.1. Définition

Dans la littérature nous retrouvons l'utilisation du terme API Economy plusieurs fois mais sans que celui-ci soit clairement défini, ou alors de manière assez ambiguë. Nous allons donc tenter d'éclaircir ce que désigne ce terme.

En premier lieu, relevons que le terme « API » dans l'API Economy est utilisé dans un contexte beaucoup plus large et englobe d'autres types d'interfaces ne pouvant pas être utilisés directement dans un langage de programmation. Il fait référence aux APIs classiques, aux Web Services et à d'autres interfaces logicielles ne faisant pas parties des deux premières catégories (tel que les « APIs software-to-human »). C'est ce qu'on appelle les « Web APIs » (APIs

conçues de manière à pouvoir être utilisées via le réseau) qui ont été le déclencheur de l'API Economy. [Gat et al., 2010]

Le terme « Economy » fait référence aux nouveaux types d'interaction métier utilisés par les entreprises. Il est habituel qu'une entreprise utilise une stratégie visant à protéger ses atouts business par rapport aux concurrents et promouvoir l'enfermement propriétaire (« vendor lock-in ») pour obtenir des avantages. L'API Economy tend à inverser la tendance: les entreprises qui auparavant mettaient leurs efforts dans la protection de leurs atouts business, vont les exposer (dans une certaine mesure) à l'aide des APIs. [Gat et al., 2010]

Avec la connaissance de la signification de ces termes, une définition peut être est proposée :

« The API Economy is the economy where companies expose their (internal) business assets or services in the form of (Web) APIs to third parties with the goal of unlocking additional business value through the creation of new asset classes. » [Gat et al., 2010]

3.1.2. Les différents acteurs

Dans le domaine de l'API Economy et dans le cadre d'une APIzation dans des modèles ouverts restrictifs et universels, il existe habituellement trois acteurs: le fournisseur API, le consommateur API et l'utilisateur final [Stylos et al., 2007; Gat et al., 2013]. Le fournisseur (ou producteur) API est une personne ou une entreprise qui fournit l'API (dans notre cas ça sera donc l'entreprise qui effectue une APIzation). Le consommateur API est une personne ou une entreprise (les développeurs tiers) qui utilise/consomme cette API pour fournir un nouveau type de produit ou service. Quant à l'utilisateur final, il s'agit d'une personne ou une entreprise qui utilise le produit ou le service que le consommateur API a conçu. Lorsque nous nous trouvons dans un modèle fermé, il est régulier que l'éditeur de logiciels qui effectue une APIzation, remplisse plusieurs rôles. Par exemple, une entreprise peut être à la fois un fournisseur et un consommateur API. Dans la littérature [Gat et al., 2013], on trouvera parfois le terme « prosumer » (qui est une combinaison des mots « producer » et « consumer ») pour rendre compte de ce rôle multiple.

3.1.3. Les avantages

Dans cette section, nous parlerons des divers avantages que l'on peut retirer de l'APIzation conjointement à l'adoption d'une API Economy. Pour ce faire, nous nous baserons largement sur un article de Gat [Gat et al., 2013] qui passe en revue ces avantages du point de vue des fournisseurs, des consommateurs et des utilisateurs finaux. Nous nous efforcerons, pour notre part, quant cela s'avère utile, de les mettre en relation avec les modèles d'APIs.

3.1.3.1. Avantages pour le fournisseur

En exposant une partie de ses activités (produits, services, etc.) via des APIS après avoir effectué une APIzation, une entreprise permettra à d'autres parties (entreprises ou personnes) de propager ces activités avec une valeur ajoutée telles que des logiciels ou des services qui seront apportées par ces parties [Gat et al., 2013]. Ceci, bien sûr, ne s'applique qu'au cas où l'APIzation s'effectue dans un modèle ouvert.

De plus, l'API Economy permettra à toutes entreprises (de la startup aux entreprises plus importantes) de pouvoir rivaliser sur le même pied d'égalité avec des entreprises déjà bien établies [Gat et al., 2013]. En effet, une petite startup pourra accéder à des ressources et services qu'elle ne possède pas via des APIs, celles-ci lui permettant de concurrencer des grandes entreprises sans devoir procéder à des investissements financiers importants.

Comme Israel Gat le relève [Gat et al., 2013], l'API Economy modifie la manière dont les marchés se forment, dont la valeur est réalisée et dont les organisations répondent. La plupart des modèles d'affaires classiques deviennent obsolètes face à l'API Economy. D'après un sondage de Vordel [Vordel, 2012], les motivations qui poussent les entreprises à utiliser et adopter les API, sont l'intégration de nouveaux canaux (50%), le déploiement des applications mobiles (25%), la construction d'une communauté de développeurs (15%), autres (10%).

Gat [Gat et al., 2013] ajoute d'autres arguments dans sa liste non exhaustive des avantages qu'apporte l'API Economy (que ce soit dans la cadre d'une APIzation dans un modèle ouvert ou fermé) : une diminution du coût et du temps de développement des logiciels produits par des tiers ou le fournisseur lui-même, un suivi de la demande d'ajout de nouvelles fonctionnalités, une couverture plus rapide et plus en largeur des divers dispositifs et plateformes, un agrandissement de la base de clients grâce à la possibilité d'établir plus aisément de nouveaux partenariats, une extension de la marque et la fidélisation.

Les APIs fournissent des possibilités de revenus très importants pour les entreprises [MuleSoft, 2015] (par exemple, Expedia génère plus de 2 milliards de dollars chaque année via les données rendues accessibles par leurs APIs). De plus, le nombre d'objets connectés à internet ne cessant de croître, il devient très intéressant pour les entreprises d'adopter l'utilisation des APIs afin de suivre la tendance au « tout connecté » et d'en retirer les bénéfices. Cet avantage nous parait d'autant plus marqué si l'APIzation s'effectue dans un modèle ouvert.

3.1.3.2. Avantages pour le consommateur

Le premier avantage pour le consommateur API porte sur la diminution du temps et de l'effort pour développer de nouveaux produits ou services [Gat et al., 2013]. Cet avantage est obtenu quel que soit le modèle dans lequel l'APIzation s'effectue, que les consommateurs API soient les développeurs tiers/partenaires, ou l'éditeur de logiciels lui-même dans le cadre d'un modèle fermé. Les APIs permettent également de ne pas se préoccuper de la manière dont la logique métier a été implémentée, les consommateurs (personnes ou entreprises) peuvent les utiliser et les exposer comme ils le souhaitent. Les consommateurs pourront combiner différentes APIs de manière créative avec une certaine facilité, produisant ainsi des produits innovants rapidement et à faible coût [Gat et al., 2013]. De plus, les APIs permettent aux consommateurs de puiser dans les bases d'utilisateurs clients des fournisseurs d'API [Gat et al., 2013]. Ces trois derniers avantages nous paraissent encore plus marqués si l'APIzation s'effectue dans le cadre d'un modèle ouvert.

La liberté dans le choix des APIs conçues par des fournisseurs d'APIs différents, constitue un avantage de poids pour le consommateur [Gat et al., 2013]. En effet, celui-ci pourra ainsi éviter plus facilement l'enfermement propriétaire (vendor lock-in). Il pourra également comparer les APIs et utiliser celles qu'il juge les mieux adaptées à ses besoins pour la création de son

produit/service. La liberté dans la comparaison et le choix des APIs ainsi que l'évitement de l'enfermement propriétaire, seront d'autant plus importants que l'APIzation sera effectuée dans un modèle ouvert universel. Dans le cadre d'un modèle ouvert restrictif, le partenaire pourra profiter d'une collaboration étroite avec le fournisseur API afin d'avoir une API qui répond au mieux à ses besoins.

3.1.3.3. Avantages communs aux deux précédents acteurs

Une entreprise qui s'implique dans l'API Economy devancera ses concurrents qui ne le font pas. Sa capacité à se développer et à évoluer sera plus importante [Gat et al., 2013].

L'avantage qu'un éditeur de logiciels a d'effectuer une APIzation (particulièrement dans un modèle ouvert universel) est qu'il se trouve en possession d'une communauté de développeurs derrière ses APIs. Cela lui permettra de pouvoir s'adapter rapidement à la demande du marché. En fin de compte, cela sera bénéfique tant aux fournisseurs et consommateurs d'APIs qu'aux utilisateurs finaux [Gat et al., 2013]. Dans le cadre d'un modèle ouvert restrictif, l'éditeur de logiciels et le partenaire (consommateur d'APIs) pourront profiter de ce nouveau canal qui est l'API afin de créer de nouvelles relations commerciales.

3.1.3.4. Avantages pour l'utilisateur final

Il est difficile d'établir tous les avantages apportés aux utilisateurs finaux puisque cette économie est encore récente [Gat et al., 2013]. Mais un avantage certain est qu'un utilisateur final disposera d'un plus large choix dans les produits et services et il pourra ainsi choisir celui/ceux qui correspondent au mieux à ses attentes.

Nous avions parlé des difficultés d'intégration d'application SaaS dans les systèmes d'information des clients. Un avantage de l'APIzation est l'utilisation des APIs, conçues par l'éditeur de logiciels pour externaliser des fonctionnalités de l'application SaaS, qui peuvent grandement faciliter l'intégration des fonctionnalités de l'application au sein de l'entreprise cliente.

3.2. Business Model

Dans cette sous-section nous commencerons par définir ce qu'est un business modèle, nous expliquerons les différents types de modèles de revenus APIs qui existent et qui pourront être choisis par l'éditeurs de logiciels lors du processus d'APIzation ainsi que les classes API qui vont faciliter le choix d'adoption d'un modèle de revenu en particulier. Dans cette sous-section nous nous focaliserons tout particulièrement sur l'APIzation effectuée dans des modèles ouverts restrictifs et universels.

3.2.1. Définition

Un business model (ou modèle d'affaire en français) est le concept qui permet à toute entreprise de gagner de l'argent. Il définit la stratégie sur laquelle l'entreprise se repose pour

faire fonctionner son activité professionnelle en garantissant une rentabilité. Il peut se formaliser dans un document de présentation de la logique globale de l'entreprise et de la manière dont elle crée de la valeur ajoutée au travers de l'exploitation d'affaires (comment elle produit de la valeur, pour qui, et comment elle gagne de l'argent). En plus de décrire l'offre que l'entreprise va proposer à ses clients ainsi que la manière dont elle va être créée et délivrée, le business model va permettre de déterminer les compétences dont l'entreprise aura besoin au sein de la société ou auprès de partenaires, de recenser les moyens matériels, immatériels, humains et financiers pour le bon fonctionnement de l'activité, de délimiter le périmètre de l'organisation en termes de production, de logistique, de gestion, etc.

Il existe énormément de business models [Zott et al., 2011] et chaque entreprise peut concevoir son propre business model en créant, adaptant, combinant des modèles déjà existants. Il est important de ne pas confondre le business model et le business plan (ou plan d'affaire en français). Ce dernier constitue la déclinaison concrète, opérationnelle et chiffrée du business model. Il est rédigé postérieurement au business model.

Voici une définition proposée par Chesbrough et Rosenbloom pour le terme business model :

« The business model is "the heuristic logic that connects technical potential with the realization of economic value" ». [Chesbrough et al., 2002]

Sur base de ressources matérielles, une entreprise pourra développer son activité et son économie. Le business model va se positionner comme intermédiaire entre ces deux domaines (figure 11).

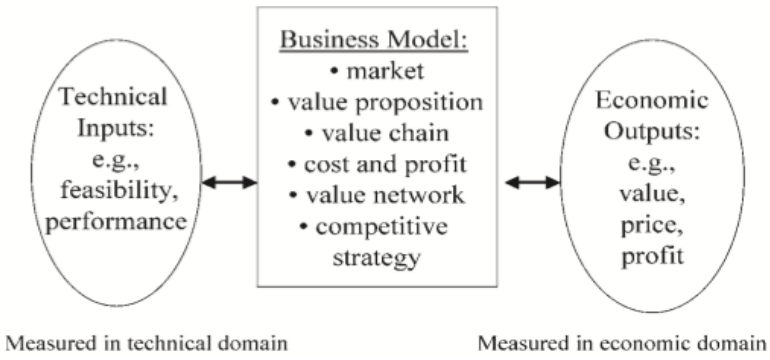

Figure 11 – Le business model est l'intermédiaire entre les domaines technique et économique. [Chesbrough et al., 2002]

Un entrepreneur débutera par l'élaboration de son business model sur base d'un travail de réflexion, de diagnostic et de synthèse. Il va ensuite rédiger son business plan qui servira à valider son business model grâce à des hypothèses et des données chiffrées.

En résumé, le business model constitue un outil qui permet d'exposer l'origine de la valeur ajoutée d'une entreprise. Il est utile pour toutes entreprises, peu importe le secteur d'activité de celles-ci. Une des composantes clés du business modèle est le modèle de revenu. Celui-ci

décrit de quelle manière l'entreprise va générer ses revenus à partir de ses produits et services. Dans la section suivante, nous allons décrire différents modèles de revenus.

3.2.2. Modèles de revenus APIs

L'éditeur de logiciels qui effectue une APIzation dans un modèle ouvert (restrictif ou universel) et qui fournit ses APIs, doit mettre en place un modèle de revenus afin de pouvoir retirer un certain avantage dans le fait de concevoir et d'exposer ses APIs. Divers auteurs [Duvander, 2011; Gat et al., 2013; Musser, 2013] ont identifié environ 20 modèles de revenus différents qui existent pour les APIs. Ces modèles de revenus sont illustrés par la figure 12.

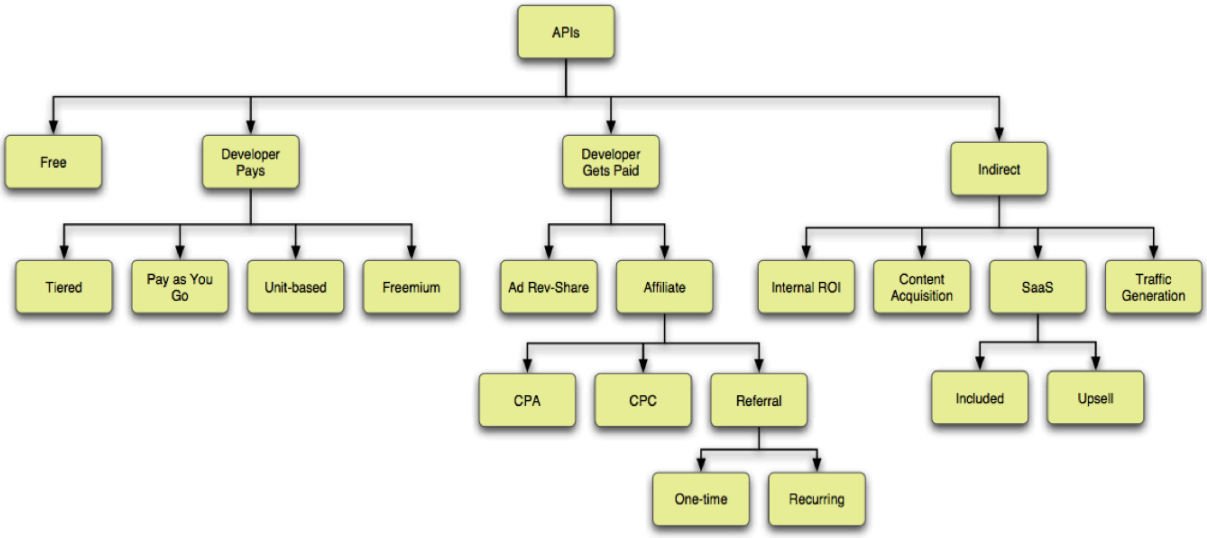

Figure 12 - Schéma reprenant les 20 modèles de revenus API [Duvander, 2011].

Ils sont catégorisés dans une structure hiérarchique par type. Quatre types principaux ont été identifiés, chacun comprennent un ou plusieurs modèles de revenus. Voici ces quatre types et leurs modèles de revenus :

3.2.2.1. Gratuit

Le fournisseur (éditeur de logiciels) met gratuitement ses APIs à disposition des consommateurs (communauté de développeurs tiers). Le but souvent recherché dans le choix d'un tel modèle est d'attirer les consommateurs ou de promouvoir l'API. Un fournisseur peut choisir d'adopter ce modèle dans un premier temps et ensuite basculer vers un autre modèle. Par exemple jusque octobre 2011, les APIs Google Maps étaient totalement gratuites, mais depuis cette date, la version gratuite est limitée à 25000 requêtes par jour et des versions premiums (payantes) ont été ajoutées.

3.2.2.2. Le consommateur paie

Les développeurs doivent payer l'API ou l'utilisation de celle-ci. Ce type de modèle comprend cinq sous-modèles [Musser, 2013 ; Gat et al., 2013] :

Paiement à l'utilisation : c'est le modèle le plus commun pour les APIs basées sur le Cloud. Les développeurs paient seulement pour les parties de l'API qu'ils utilisent. Cela permet aux développeurs d'avoir une offre adaptée à leurs besoins.

Freemium : dans ce modèle, la version de base de l'API est distribuée gratuitement, mais elle contient certaines limitations. Une autre version est mise à disposition -il s'agit de la version premium- qui lève les limitations de la version de base mais le consommateur doit dans ce cas payer pour pouvoir l'utiliser. Les développeurs peuvent changer de version à n'importe quel moment.

À l'unité : les développeurs paient en fonction du nombre d'appels/requêtes effectués à l'API. Habituellement ces requêtes sont regroupées en unités (par exemple par type de requêtes) et sont facturées par paliers (exemple : pour 500 requêtes API).

Graduellement/par niveau : le fournisseur API propose des niveaux d'utilisation de son API. Les niveaux sont prédéfinis avec certaines caractéristiques telles que le nombre total d'appels par mois, le nombre de ressources du fournisseur mises à disposition, etc. Les développeurs paient donc en fonction du niveau choisi.

Frais de transaction (sous-modèle non repris dans la figure 12): le fournisseur API prélève un montant sur chaque transaction réalisée via l'API. Par exemple pour effectuer des paiements via des systèmes bancaires (exemple : PayPal).

3.2.2.3. Le consommateur est payé

Les développeurs sont payés par le fournisseur d'API. Ce type de modèle comprend deux sous-modèles [Musser, 2013 ; Gat et al., 2013] :

Revenu partagé : le fournisseur partage un pourcentage sur le revenu gagné grâce à l'utilisation de l'API avec le consommateur API (par exemple avec un revenu basé sur la publicité).

Affiliation : le fournisseur paie le consommateur sur base d'un contrat. La plupart du temps ce modèle est utilisé avec de la publicité (par exemple : coût par action/acquisition (CPA), coût par clic (CPC) ou coût par mille (CPM, concrètement 2€CPM signifie que l'annonceur doit payer 2 € chaque fois que 1000 affichages de publicités sont effectués)) [Investiopedia, 2015] ou avec un programme pour amener de nouveaux clients (par exemple, pour chaque contact invité devenu client, une offre de réduction est proposée). Le fournisseur peut mettre en place un système de rémunération effectué à un seul moment (par exemple, paiement pour chaque nouveau client lors de son inscription) ou de manière récurrente (par exemple, paiement mensuel pour les clients qui restent souscrits).

3.2.2.4. Indirect

Le fournisseur utilise des méthodes indirectes de paiements ou de revenus. Ce type de modèle comprend quatre sous-modèles [Musser, 2013 ; Gat et al., 2013] :

Acquisition de contenu : dans ce cas, le revenu s'effectuera sur base de l'utilisation de la collecte des données des utilisateurs utilisant l'API (par exemple : annonces, messages, commentaires, opinions, etc.).

SaaS : lorsque les APIs font partie d'un business modèle SaaS, elles peuvent être fournies soit dans l'offre SaaS soit comme service supplémentaire (add-on). Par exemple, Salesforce offre des APIs d'accès à ses plates-formes, mais uniquement aux sociétés qui paient la licence entreprise. Salesforce sait que l'intégration d'APIs est importante et l'utilise pour inciter les clients à souscrire des abonnements plus chers.

Dissémination du contenu : dans le but d'obtenir une exposition plus large, le fournisseur créera du contenu (news, articles) et le rendra disponible à des parties tierces afin que celles-ci puissent les utiliser et les partager. Par exemple, le New York Times qui a beaucoup de contenu utilise des APIs pour distribuer le contenu vers ses partenaires.

Utilisation interne : ce modèle de revenus est propre au modèle d'APIs fermé. Le fournisseur concevra des APIs dans le but d'une utilisation en interne. Ces APIs peuvent être utilisées pour rendre accessible une fonctionnalité depuis un appareil mobile ou un site web de l'éditeur.

Ceci représente bien sûr une liste non exhaustive. De plus, il est tout à fait possible de combiner différents modèles de revenus pour une API. La tendance actuelle est de créer des APIs plus pour le partage de données que pour offrir une fonctionnalité ou un service [CCSK Guide, 2012]. Pour faciliter le choix du ou des modèles de revenus à choisir lors du processus d'APIzation, il peut être utile de comprendre ce qu'on appelle les classes API.

3.2.3. Les classes API

La classe dépend du rôle fonctionnel principal de l'API en question. Guillaume Balas en a défini quatre [Balas, 2011] :

1) L'API est le produit :
   - Revenu direct
   - Paiement par transaction
   - Tarification par niveaux

Dans ce cas-ci, l'éditeur de logiciels propose uniquement ses APIs comme « produit » aux développeurs tiers/partenaires et rien d'autre.

2) L'API projette le produit :
   - Fournir plus d'utilité
   - Permettre un accès depuis des périphériques mobiles
   - Permettre une intégration en profondeur

Ici, l'API est liée à un produit déjà existant (par exemple une application SaaS). L'objectif de cette API est de « projeter » le produit c'est-à-dire qu'elle va améliorer/faciliter/étendre son utilisation. Par exemple, Google a dû créer des APIs pour son application SaaS Gmail afin de pouvoir accéder aux fonctionnalités de cette application depuis une application mobile native.

3) L'API favorise le produit :
    - Acquisition de nouveaux utilisateurs
    - Publicités
    - Promotion de la marque
    - Programmes d'affiliation

Ce type d'APIs va « favoriser » un produit déjà existant dans ce sens que ces APIs sont conçues dans le but de faire connaître le produit pour acquérir de nouveaux clients. Typiquement, ce type d'API peut être utilisée pour afficher un bandeau publicitaire dans une application mobile afin de faire connaître un produit à l'utilisateur de l'application.

4) L'API nourrit le produit :
    - Acquisition de contenus
    - Lier les partenaires
    - Innovation interne

Dans cette classe, l'API sert à « nourrir » un produit existant. Elle va permettre d'apporter des données que le produit va utiliser. Ces données peuvent par exemple être du contenu (texte descriptif à propos de quelque chose), les informations des utilisateurs, des commentaires, des notes d'évaluation (rating), etc. Un exemple concret est l'API que Facebook met à disposition pour pouvoir ajouter du contenu (photos, vidéo, commentaires) à partir d'un site web ou d'une application mobile.

Les deux premières classes sont basées sur un revenu direct tandis que les classes trois et quatre sont basées sur un revenu indirect. Ces classes vont permettre aux fournisseurs de savoir vers quel modèle de revenu API ils peuvent se diriger en se basant sur le rôle que leurs APIs auront (être le produit, projeter le produit, favoriser le produit, ou nourrir le produit). [Balas, 2011]

3.3. Business Model Canvas

Il existe plusieurs canevas/format pouvant être utilisés pour représenter un business modèle d'une organisation. Parmi ceux-ci, nous pouvons citer, à partir de la revue effectuée par Arbache [Arbache, 2012], les business modèles de M.W Johnson (quatre composants), de Z. Lindgart et M. Reeves (BCG, six composants) et de A. Osterwalder et Y. Prigneur (neuf composants). Pour la suite de ce mémoire, nous retiendrons le business model d'Alexander Osterwalder car il constitue le modèle le plus utilisé actuellement et « est clair, simple, explicite […] particulièrement adapté aux startups et entrepreneurs […] favorisant la créativité…» [Arbache, 2012].

Ce canevas [Osterwalder et al., 2011] a pour but principal d'être compréhensible par tout le monde afin de pouvoir faciliter la description et la discussion entre les différents acteurs. Bien que simple, pertinent et compréhensible, ce canevas ne simplifie pas toute la complexité

derrière le fonctionnement d'une organisation. Il permet de représenter en une seule page, au travers d'un canevas, l'ensemble du modèle économique de l'entreprise. Il est composé de neuf blocs qui couvrent les quatre principaux domaines d'une entreprise : les clients, l'offre, l'infrastructure, la viabilité financière.

Dans la sous-section suivante, nous établirons les changements qu'une démarche d'APIzation apporte dans une entreprise au moyen du business model canvas. Une description des neuf blocs du business model canvas est disponible en annexe.

3.4. Business model canvas d'une APIzation

Nous allons passer en revue les blocs dans lesquels il est susceptible de survenir une modification suite à la mise en œuvre d'une APIzation dans un modèle ouvert au sein d'une entreprise éditrice de logiciels. Nous détaillerons dans certains blocs les différences qui peuvent avoir lieu en fonction du type de modèle ouvert d'APIs.

3.4.1. Le segment de clientèle

Lorsqu'une entreprise effectue une APIzation, elle spécifiera le segment de clientèle qu'elle ciblera. Dans le cadre d'un modèle ouvert universel, elle visera tous les développeurs externes à l'entreprise. Ceux-ci peuvent être des développeurs indépendants ou faisant partie d'une organisation et cherchant à intégrer les APIs dans leurs solutions pour créer de nouveaux produits ou services. Dans le cadre d'un modèle ouvert restrictif, le segment de clientèle sera fortement restreint. Ce segment sera celui dont font partie les partenaires avec lesquels on souhaite distribuer l'API. Il est peut-être utile, dans une certaine mesure, de spécifier le segment de clientèle pour les utilisateurs finaux. Par exemple : une entreprise pourrait cibler des utilisateurs finaux indépendants ou travaillant dans une entreprise dans un domaine particulier (assurance, médical, sportif etc.).

3.4.2. La proposition de valeur

Pour un modèle ouvert (restrictif ou universel), la proposition de valeur portera principalement sur les fonctionnalités qu'offre l'entreprise par l'intermédiaire de son API. En plus de ces fonctionnalités, il est possible d'ajouter divers éléments qui vont apporter de la valeur ajoutée à l'API. Parmi ces éléments, il est possible de citer les différents attributs de la qualité du service (QoS : débit, disponibilité, gigue, taux de perte de paquets, etc.) qui seront renseignés dans le SLA (Service-Level Agreement), la sécurité, la documentation de l'API, le support de divers formats (JSON, XML), la facilité d'utilisation. Tous ces éléments sont très importants dans le cadre d'un modèle ouvert étant donné que les APIs sont destinées à être utilisées par des développeurs extérieurs. Ces éléments vont contribuer fortement à la viabilité du business model. Pour un modèle ouvert restrictif, le support apporté au(x) partenaire(s) sera de haute qualité et également personnalisé en fonction de ses/leurs besoins. Il est également possible que l'éditeur de logiciels et ses partenaires signent un contrat d'exclusivité.

3.4.3. Les canaux

Dans le cas d'une APIzation dans un modèle ouvert, un nouveau canal important est ajouté, c'est l'API. De plus, dans le cadre d'un modèle ouvert universel, il est possible que le concepteur d'APIs mette en place un site web afin de communiquer plus largement et plus en détail à propos de son offre et la distribuer, éventuellement de manière automatisée. Dans le cadre d'un modèle ouvert restrictif, le site web servira à faire connaître la marque et les services proposés. Afin d'acquérir de nouveaux partenaires, l'éditeur de logiciels devra s'impliquer dans une gestion marketing (ex. : prospection chez les potentiels futurs partenaires). Le support pourra être effectué via emails ou un système de messagerie instantanée.

3.4.4. Les relations clients

Les relations clients dépendront bien évidemment du segment de clientèle que l'entreprise souhaite viser. Lors d'une APIzation dans un modèle ouvert universel, l'entreprise peut opter pour un self-service via un site web qui permettra aux développeurs externes (consommateurs des APIs) de s'enregistrer automatiquement, de choisir leur plan tarifaire lié à l'utilisation de l'API, de récupérer les informations de sécurité leur donnant accès à l'API. Il faudra bien sûr pouvoir fournir un support. En ce qui concerne les utilisateurs finaux, le fournisseur API pourra jouer sur son image de marque. Par exemple dans le cadre d'un partenariat, référencer les partenaires permet d'ajouter du poids aux produits face aux solutions concurrentes. Dans le cas d'un modèle ouvert restrictif, une assistance personnalisée (communication avec des experts, dépannage immédiat, etc.) est souvent mise à disposition des partenaires.

3.4.5. Les flux de revenus

Les flux de revenus dépendront des modèles de revenus que le fournisseur choisira de mettre en place. Dans une section précédente, nous avons abordé différents moyens existants afin qu'une entreprise puisse retirer un gain (direct/indirect, ponctuel/récurrent). Le fournisseur devra, dans ce bloc, expliciter le(s) type(s) de modèles de revenus API qu'il mettra en place. Dans le cadre d'une APIzation dans un modèle ouvert (restrictif ou universel), le fournisseur aura le choix entre les modèles direct et indirect. On notera qu'il est important de spécifier dans ce bloc la fréquence de revenus (ponctuel/récurrent) adoptée, car elle pourrait avoir un impact plus ou moins important sur la stratégie. Il n'y a pas de modèles de revenus spécifiques à un sous-modèle ouvert d'APIs. Ceux-ci peuvent convenir à la fois pour un modèle ouvert restrictif et un modèle ouvert universel. On soulèvera toutefois une tendance actuelle qui consiste, pour un modèle ouvert universel, de fournir un service de base limité gratuitement et de proposer des packs premiums payants soulevant les limitations imposées avec la version gratuite.

3.4.6. Les ressources clés

Parmi les ressources clés d'un éditeur de logiciels pour effectuer une APIzation dans un modèle ouvert, on citera bien sûr les développeurs internes qui concevront l'API ainsi que l'application source qui comprend les fonctionnalités qui doivent être externalisées et l'infrastructure (matériels et logiciels) mise en place pour distribuer les APIs. Dans le cas d'un

modèle ouvert universel, la communauté de développeurs peut jouer un rôle important dans l'amélioration de l'API et peut donc constituer une ressource clé à ce niveau. Lorsqu'il s'agit d'un modèle ouvert, l'équipe de support est très importante pour fournir un service après-vente de qualité.

3.4.7. Les activités clés

Ce bloc contient l'activité d'APIzation ainsi que les activités permettant de gérer la plateforme APIs (création, maintenance, évolution), la formation continue de l'équipe de développements afin d'adapter les APIs aux nouvelles technologies et formats de données. Dans le cas d'un modèle ouvert, le support constitue également une activité clé afin d'aider les développeurs externes à pouvoir les utiliser (ce point est tout particulièrement important lors d'un partenariat pour un modèle ouvert restrictif).

3.4.8. Les partenaires clés

Que ce soit dans un modèle ouvert restrictif ou universel, un partenariat peut être effectué avec des entreprises proposant des plateformes de gestion d'APIs [Raivio et al., 2011] qui comprennent des outils d'analyse (par exemple, AmberPoint, Apigee, Denodo, DirectAPI. Voir la liste en annexe 6.1.). Dans ce cas, on externalise toute la plateforme APIs et la gestion de celle-ci se fait à distance via les outils proposés par l'entreprise. Il s'agit d'une solution fort intéressante pour les entreprises n'ayant pas les moyens ou ne souhaitant pas prendre en charge le coût de développement et de maintenance d'une telle plateforme. Il peut s'agir de startups, mais aussi de grandes entreprises (par exemple, Apigee compte parmi ses clients des entreprises telles qu'Adobe, AT&T, Dell, eBay, etc.).

3.4.9. La structure des coûts

L'APIzation peut apporter des coûts supplémentaires. Ces coûts seront liés au projet d'APIzation en lui même (l'équipe du projet, la création de l'API, la partie marketing pour la faire connaître) mais aussi au bon fonctionnement sur le long terme (la maintenance et l'évolution de l'API, la gestion de son cycle de vie). De plus, le fournisseur peut choisir un modèle de revenus API qui consisterait à rémunérer les utilisateurs de l'API, ce qui impliquera encore des coûts supplémentaires. Une augmentation de coûts est également à envisager si l'APIzation conduit à faire appel à une entreprise externe qui s'occupera de gérer la plateforme APIs (cette entreprise devient alors un partenaire) [Raivio et al., 2011]. Le support client dans le cadre d'une APIzation dans un modèle ouvert restrictif, constitue aussi une source importante de dépenses.

Les figures 13 et 14 sur les pages suivantes présentent chacune un business model canvas synthétisant les différents points que nous venons de discuter. La première figure se focalise sur le modèle ouvert restrictif et la seconde sur le modèle ouvert universel.

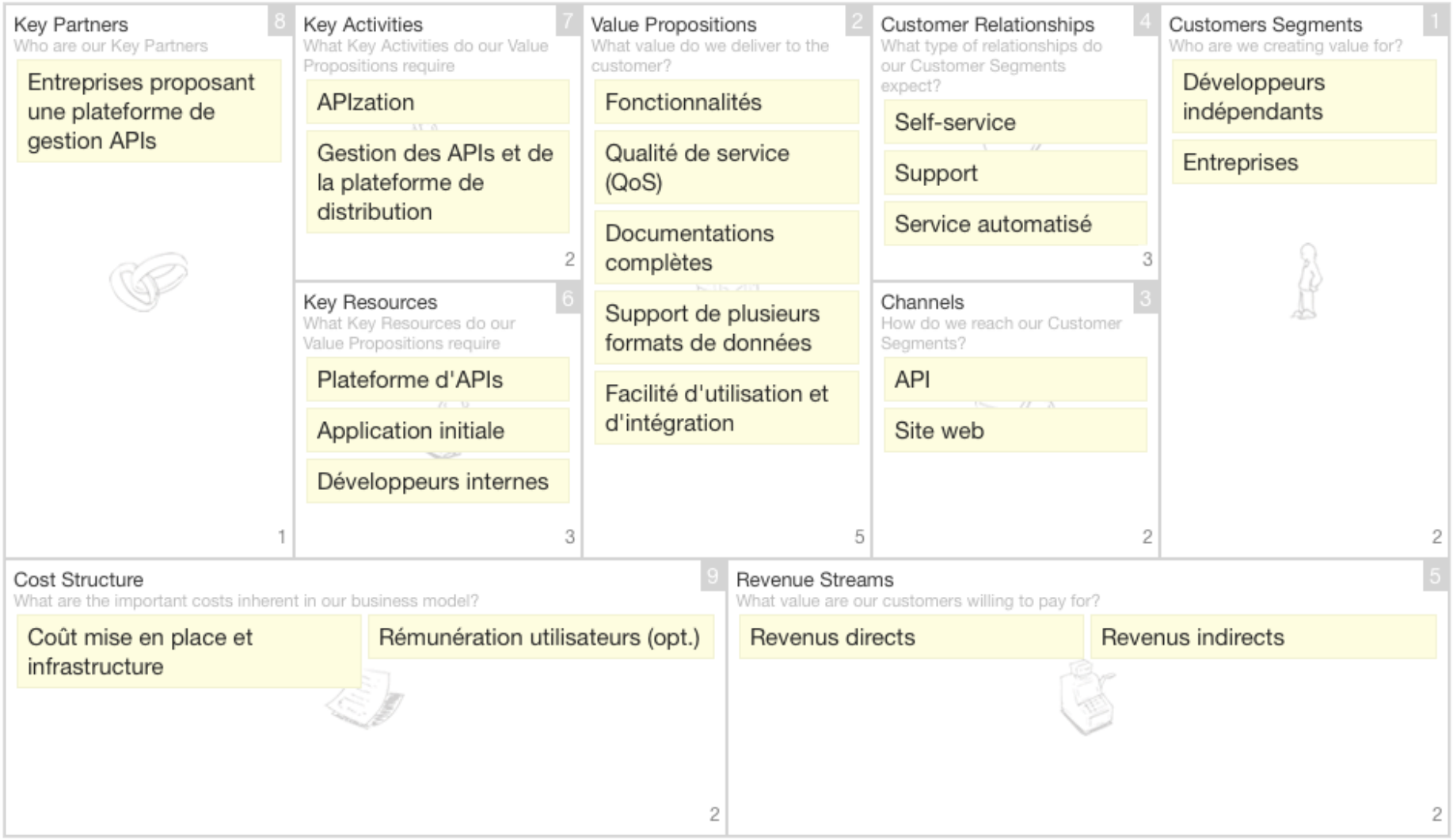

Figure 13 – Business model canvas d'un éditeur de logiciels effectuant une APIzation dans un modèle ouvert universel.

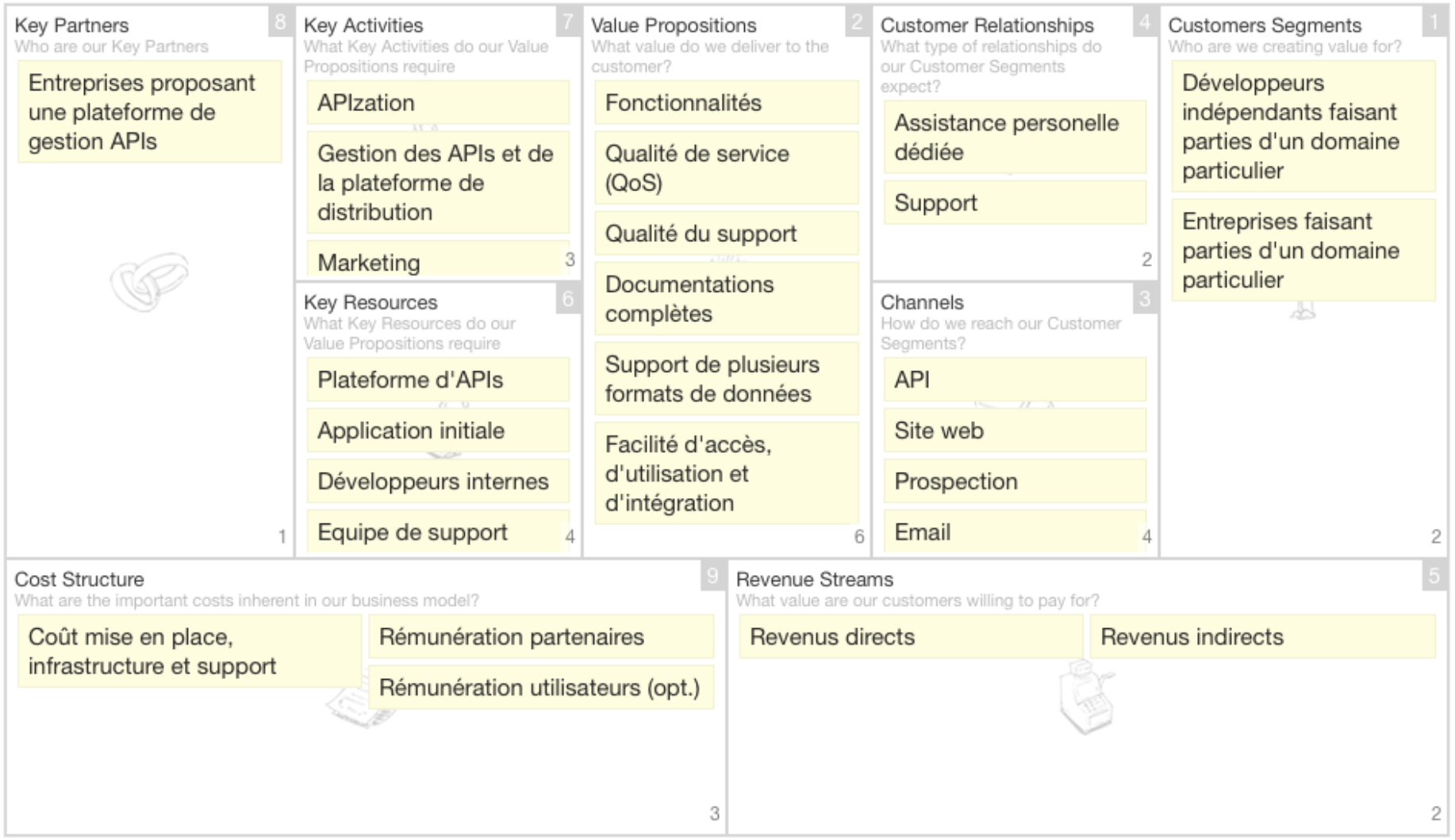

Figure 14 – Business model canvas d'un éditeur de logiciels effectuant une APIzation dans un modèle ouvert restrictif.

4

MÉTHODE D'APIZATION

Après avoir défini ses besoins et établi/adapté son business model, l'éditeur de logiciels peut commencer à concevoir ses APIs. Ceci constituera l'objet de cette section. Nous commencerons en abordant les défis auxquels l'entreprise doit faire face lors d'une APIzation. Ensuite, nous proposerons une méthodologie pour effectuer une APIzation. Tout d'abord, nous déterminerons quelle approche est la plus adéquate. Ensuite, nous expliquerons une méthode pour aider l'éditeur de logiciels à identifier quelles sont les fonctionnalités qui doivent être externalisées. Après cela, nous détaillerons l'étape clé qui consiste à concevoir les APIs en elles-mêmes et les technologies les plus couramment utilisées actuellement. Pour finir, nous nous attarderons sur un autre point important qui est la livraison/distribution des APIs. Pour y procéder, il faudra choisir un style architectural. Étant donné qu'il existe parfois une confusion entre SaaS et SOA, nous expliquerons les différences entre ceux-ci avant de poursuivre l'explication des étapes de la méthodologie. Tout au long de cette section, nous expliquerons une démarche à suivre ainsi que les points essentiels à prendre en compte pour mener à bien les différentes étapes d'une APIzation.

4.1. Défis à surmonter

L'implication d'une économie API inclut également divers défis que le fournisseur doit gérer au mieux [Ajami, 2015; Gat et al., 2013]. Afin de minimiser les chances que surgissent ces contrariétés lors d'une étape du développement ou lors de la phase d'utilisation de l'API, il est préférable que le fournisseur en ait connaissance et se prépare correctement dès le début du projet. À ces défis décrits dans la littérature, nous en avons ajouté deux autres que nous avons également identifié (la scalabilité et le contrôle). Les défis sont listés ci-dessous. Nous avons également à chaque fois discuté des nuances à y apporter en fonction du modèle ouvert adopté lors de l'APIzation.

Sécurité et confidentialité : En raison de l'ouverture et du partage des données avec des parties tiers, les risques en matière de sécurité sont augmentés. Les problèmes de confidentialité devront également être pris en compte. Il sera donc nécessaire d'utiliser des technologies et solutions liées à la sécurité (types d'authentification, autorisation des utilisateurs, gestion des identités, etc.). Un éditeur de logiciels qui effectue une APIzation en choisissant un modèle ouvert universel devra particulièrement protéger l'accès et l'utilisation de ses APIs car tout le monde peut prendre connaissance de l'existence de ces APIs. Il faut donc qu'elles soient seulement utilisables par des développeurs enregistrés et ayant souscrit à une formule d'utilisation. Un haut niveau de sécurité est également important dans le cadre d'un modèle ouvert restrictif pour autoriser l'utilisation des APIs seulement par les partenaires

et afin de sécuriser le canal de communication dans le cas de transmission de données sensibles.

Acquérir et encadrer les consommateurs APIs : Le fournisseur APIs devra aussi attirer les consommateurs grâce à une stratégie API, une bonne gestion de l'écosystème API et une mise en place d'un API marketing. Lors d'une APIzation dans un modèle ouvert universel, il est très important de développer et d'encadrer correctement la communauté de développeurs qu'il y aura autour de l'API. Les moyens existants pour encadrer cette communauté sont de mettre à disposition des développeurs tiers un site internet où ils pourront trouver tous les « changelogs » des APIs et contribuer à l'amélioration des APIs en faisant part de leurs suggestions. Dans un modèle ouvert restrictif, le fournisseur d'APIs doit se concentrer uniquement sur les partenaires. Il devra rechercher des partenaires via des moyens de prospections. Pour fournir un encadrement adéquat à ses partenaires, le fournisseur APIs doit établir une bonne communication en leur fournissant tous les détails techniques et le support nécessaire.

Confiance entre les fournisseurs et les consommateurs : Que ce soit dans un modèle ouvert restrictif ou universel, le fournisseur doit pouvoir établir et maintenir la fiabilité entre les APIs exposées et la confiance avec le consommateur. L'évolution de l'API est inévitable et même bénéfique, mais il est très important que le fournisseur se préoccupe du maintien de la conformité de l'API ainsi que d'une compatibilité ascendante. Lorsqu'il doit introduire des changements dans ses APIs, le fournisseur se doit d'être vigilant, car cette action pourrait engendrer un risque de rupture avec le consommateur (par exemple : supprimer le service, créer une nouvelle version incompatible avec la précédente, changer les conditions d'utilisation, créer des produits similaires à ceux offerts par le consommateur API, etc.). Dans le cadre d'une APIzation dans un modèle restrictif, la rupture s'effectuera sur un petit nombre de consommateurs APIs (les partenaires) mais qui sont cruciaux. Tandis que dans un modèle ouvert universel, la rupture risque de s'effectuer avec un nombre beaucoup plus élevé de consommateurs APIs (tous les développeurs tiers). Il est donc conseillé d'effectuer une analyse des répercussions qui peuvent avoir lieu quand des modifications sont envisagées.

Qualité API, réussir la création de bonnes APIs : Pour mener au mieux son processus d'APIzation et donc son projet de création d'APIs, le fournisseur devra se soucier de créer des APIs efficaces (assurer une qualité de service d'un certain niveau) et efficientes. Lorsque le modèle de revenu est payant pour le consommateur des APIs (modèle ouvert restrictif ou universel), il est essentiel d'assurer une qualité correcte et de fournir exactement les fonctionnalités prévues (les consommateurs paient pour un service).

Nombres d'APIs : Lorsqu'il y a trop d'APIs proposées, il devient difficile pour le consommateur de choisir l'API qui lui convient. Il est donc nécessaire de bien définir les fonctionnalités qui peuvent être rassemblées derrière une API afin de ne pas multiplier le nombre d'APIs potentiellement semblables. Cela est surtout valable lors d'une APIzation dans un modèle ouvert universel. Dans un modèle ouvert restrictif, le fournisseur APIs sera en mesure de proposer l'API qui convient le mieux à ses partenaires, car il analysera chacun de ses besoins et concevra une API spécifique (ou adaptera une API générique) pour ce partenaire.

Documentations des APIs : Certaines APIs peuvent être plus difficilement intégrables dans un programme ou SI, aussi le fournisseur devra s'efforcer de faciliter l'étape d'intégration de ses APIs. Pour ce faire il devra fournir une documentation correcte et complète.

Défis techniques : parmi les autres défis techniques qui peuvent surgir lors d'une APIzation, citons notamment la gestion des données et des APIs, surtout lorsque la quantité est forte importante (il faut trouver une méthode rentable d'accès de partage aux données). L'incompréhension ou la mauvaise utilisation des technologies (REST, SOAP, etc.) peuvent également être considérées comme des défis.

Nous pouvons compléter cette liste de défis, en y ajoutant celui de la scalabilité et du contrôle.

Scalabilité : lorsqu'un fournisseur APIs souhaite passer d'un modèle fermé vers un modèle ouvert ou d'un modèle ouvert restrictif vers un modèle ouvert universel, il devra effectuer plusieurs changements qui peuvent être délicats. Tout d'abord il devra très probablement modifier son business model et s'assurer de sa viabilité. La documentation devra également être adaptée aux futurs lecteurs. Les outils de « metering » (par exemple d'analyse statistique d'utilisation) devront également être adaptés pour prendre en charge un nombre plus important d'utilisateurs APIs et de requêtes.

Contrôle : le fournisseur APIs devra également mettre en place une gestion de contrôle efficace de ses APIs afin de ne pas perdre du temps inutilement et ne pas faire d'erreurs. Pour cela il doit prendre en compte le cycle de vie des APIs et établir une stratégie pour contrôler l'ensemble de ses APIs (versions, accessibilité, etc.).

4.2. Méthodologie proposée

Dans cette sous-section, nous abordons les différentes étapes de la méthodologie proposée afin d'effectuer une APIzation. Cette méthodologie est le résultat d'une comparaison, fusion et adaptation de plusieurs parties de méthodes trouvées dans la littérature.

4.2.1. Approches existantes

Nous trouvons dans la littérature [Arsanjani, 2004; Inaganti et al., 2007; MuleSoft, 2015] deux principales approches qu'il est possible d'adopter pour concevoir des nouveaux services informatiques. Ces approches sont qualifiées de « Top-Down » (descendante en français) et de « Bottom-Up » (ascendante en français). Dans les sous-sections suivantes, nous décrirons ces deux approches. Nous choisirons ensuite celle qui convient le mieux dans le cadre d'un processus d'APIzation et nous argumenterons ce choix.

4.2.1.1. Approche Top-Down

Cette approche peut être divisée en deux types : une approche axée sur les processus métier et une approche axée sur les cas d'utilisation [Arsanjani, 2004 ; Inaganti et al., 2007]

Dans le cas d'une initiative visant à concevoir un service, les futures fonctionnalités devraient être conduites par une analyse en profondeur de la chaîne de valeur et des processus métier. Si

l'initiative est spécifique à un projet et que l'entreprise n'a pas d'expérience quant à l'adoption d'un type particulier d'architecture, la plupart du temps ce projet sera conduit par les cas d'utilisation (use cases). Dans les deux cas, il est quand même conseillé d'apporter le point de vue des processus métier afin d'éviter le danger de construire un logiciel avec une architecture cloisonnée (« stove-piped », « système ou méthode développée en isolation, sans considération quant à son fonctionnement avec les technologies existantes ou futures » [Rouse, 2015]).

4.2.1.1.1. Approche axée sur les processus métier (analyse de la chaine de valeur)

La figure 15 illustre le type de schéma utilisé lors de l'étape d'identification des activités métiers de l'approche « Top-Down ».

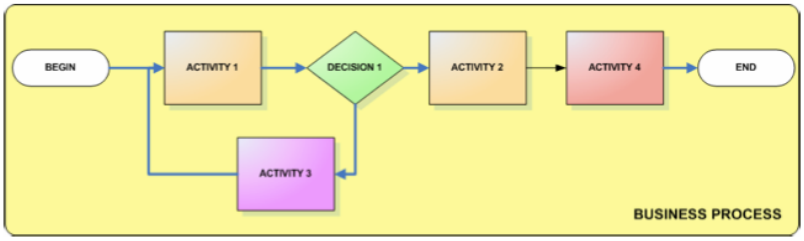

Figure 15 – Approche « Top-Down »: identification des activités métiers [Inaganti et al., 2007]

Ce type d'approche top-down consiste à effectuer une analyse de la chaine de valeur d'un point de vue IT et business. Cette analyse apportera aux consultants et architectes une meilleure compréhension de l'entreprise, de ses fonctions critiques, non critiques et de supports. L'attention est attirée sur l'importance de posséder une compréhension claire des différentes interactions entre les nombreuses fonctions pour concevoir les modules qui vont offrirent les nouveaux services.

Voici les principales activités de cette approche [Inaganti et al., 2007] :

- Comprendre et capturer les grands domaines fonctionnels et pouvoir comprendre les interactions qui existent entre chacun d'entre eux.
- Détailler les processus métier et les points de contact entre eux (typiquement en utilisant un diagramme avec la notation BPMN [Dijkman et al., 2008]).
- Détailler les sous-processus.
- En se basant sur les processus d'activités, identifier les services candidats de haut niveau.
- Comprendre les limites du système informatique existant.

Une bonne documentation des processus métier est importante, car elle aidera les architectes à comprendre le contexte métier, relier les domaines business et IT, lister les activités à effectuer

en premier, etc.

4.2.1.1.2. Approche axée sur les cas d'utilisation

Dans ce type d'approche top-down [Inaganti et al., 2007], l'équipe de conception commencera par les cas d'utilisations (use cases) dont les processus métier ne sont pas complets ou disponibles. Même si ceux-ci sont disponibles, la tendance est de commencer avec la documentation des cas d'utilisation et de suivre le cycle de vie du processus de développement logiciel conventionnel. Une liste des services sera déterminée pour les scénarios fonctionnels communs et pour l'identification des sous-systèmes.

Quelle que soit l'approche suivie (axée sur les processus métier ou axée sur les cas d'utilisation), il est fortement conseillé de structurer l'équipe travaillant sur le projet en deux sous équipes [Inaganti et al., 2007] : une équipe des services qui se chargera d'essayer d'identifier les exigences de base des services, et une équipe d'application qui devra comprendre les exigences de chaque sous-système/module et qui transmettra à l'équipe des services leurs attentes pour chaque service dans chacun des modules. Cette collaboration conduira à un document de spécifications pour chacun des services.

Cette approche axée sur les cas d'utilisation comprend les activités de développement utilisées pour la conception de système informatique conventionnel (analyse des sous-systèmes, modélisation des cas d'utilisation, élaboration des spécifications des composants). Elle comprendra également les activités qui sont nécessaires pour la conception des services : liaison entre service et composant (cette activité englobe l'identification des responsabilités des services et l'attribution des responsabilités des services aux composants), l'élaboration des spécifications des services qui englobent les détails des interfaces, l'utilisation de méthodologies et toutes les interactions des cas d'utilisation métier (cette dernière sous-activité est importante, car on pourra concevoir des services génériques et durables).

Dans le but d'affiner les services et leurs responsabilités, les services identifiés avec cette approche devront être mis en corrélation avec la liste des activités identifiées avec l'approche « top-down » axée sur les processus métier [Inaganti et al., 2008] (figure 16).

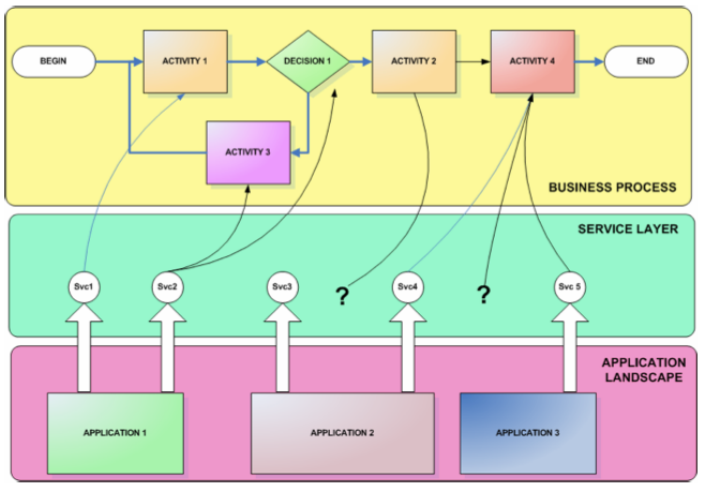

Figure 16 – Corrélation entre les activités métiers et les services [Inaganti et al., 2008]

4.2.1.2. Approche Bottom-Up

La figure 17 schématise l'approche Bottom-up [Arsanjani, 2004; Inaganti et al., 2007; MuleSoft, 2015] .

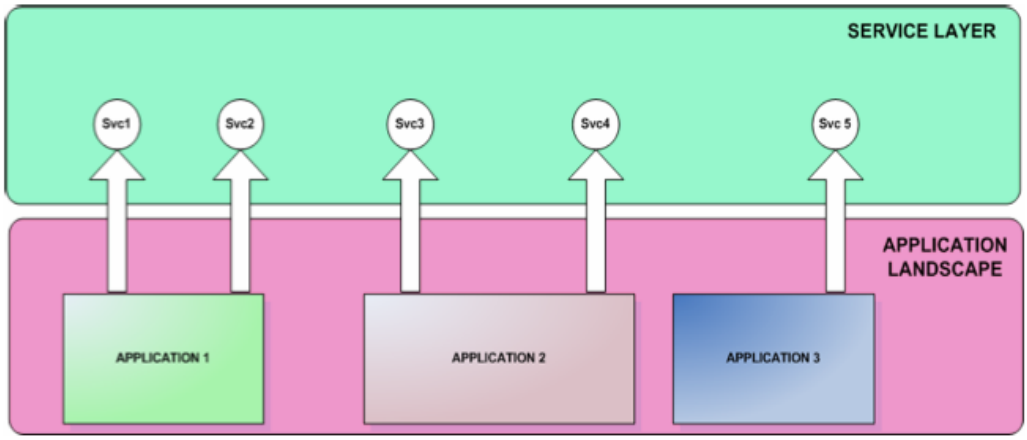

Figure 17 – Approche « Bottom-Up » : les fonctionnalités offertes par les applications sont exposées comme services [Inaganti et al., 2007]

Cette approche consiste à utiliser l'analyse des fonctionnalités du portefeuille d'application de l'entreprise, l'évaluation de la réutilisation, de la redondance et l'effort de rationalisation. Elle vise directement la redondance de la logique métier, les multiples copies de données ou l'implémentation multiple de logique métier pour différents produits, ce qui conduit à un coût élevé de l'exploitation et de la maintenance de ces produits. Un exemple [Inaganti et al., 2007] d'implémentation d'une même logique métier pour plusieurs produits pourrait être qu'une société souhaite implémenter un système de recherche en utilisant différents produits tels que Autonomy, Verity, Google. Dans cet exemple, le but commun des différents produits consiste à effectuer une recherche. On essaiera donc d'implémenter une même logique métier pour ces différents produits afin d'éviter une quelconque redondance et au final d'éviter les coûts engendrés par cette redondance.

On notera qu'une composition des services identifiés via cette approche n'est pas obligatoire pour construire les processus métier, bien qu'une composition de ces services puisse avoir un impact plus ou moins élevé en fonction du scénario construit. Ces services feront partie intégrante des services de base tels que les services d'infrastructures ou techniques.

4.2.2. Approche adoptée

Le processus d'APIzation ayant pour but d'externaliser les fonctionnalités d'une application informatique déjà existante, l'approche qui convient le mieux est l'approche « Bottom-Up » (ascendante). Nous suivrons donc cette approche en commençant par analyser l'application existante pour identifier les fonctionnalités à externaliser. Après, des APIs seront conçues pour accéder à ces fonctionnalités et, finalement, ces APIs seront distribuées à la manière de service aux consommateurs. Ces différentes étapes sont détaillées dans les sous-sections suivantes.

4.3. Identification des fonctionnalités à externaliser

L'étape qui consiste à déterminer les fonctionnalités qui devront être externalisées est fort importante. En effet, créer des APIs pour toutes les fonctionnalités d'une application constitue un travail conséquent alors qu'elle peut ne pas être nécessaire. De plus, un mauvais choix des fonctionnalités au départ aura des répercussions sur le reste du projet ainsi que sur le ROI des APIs. Sur base de plusieurs méthodologies trouvées dans la littérature [Almonaies et al., 2011 ; Nassif et al., 2010; Sneed, 2006], il est possible d'adopter une démarche progressive et efficace pour identifier ces fonctionnalités.

En premier lieu, il convient d'analyser l'application dans son ensemble et déterminer les parties pour lesquelles il serait utile d'effectuer une démarche d'APIzation [Almonaies et al., 2011 ; Nassif et al., 2010; Sneed, 2006]. Un éditeur de logiciels, sur base d'outils de surveillance, peut déterminer quelles parties sont les plus utilisées par ses utilisateurs (par exemple l'agenda, la traduction automatique, etc.). Elle pourrait aussi déterminer les parties de l'application utiles à externaliser en réfléchissant sur les fonctionnalités qu'elles contiennent et identifier celles qui seraient susceptibles d'être compétitives sur le marché.

Une fois les parties identifiées, il convient de les étudier et les analyser soigneusement afin d'identifier leurs spécifications (niveau de maturité, langage de programmation, taille, etc.) dans le but de pouvoir déterminer le niveau de dépendance entre les fonctionnalités (autrement dit, le niveau de couplage entre celles-ci).

Ensuite, sur base des parties de l'application et de leurs spécifications, la fonction et la valeur commerciale de chaque fonctionnalité candidate seront identifiées [Nassif et al., 2010; Sneed, 2006 ; Almonaies et al., 2006]. Pour cela, l'éditeur de logiciels peut se baser sur les règles business contenues dans les différentes parties de l'application [Sneed, 2006]. Bien souvent, une fonctionnalité contient une seule règle business complexe. Quand une règle business complexe est identifiée, il est nécessaire de la découper en règles business élémentaires qui constituent des unités logiques autonomes. Par exemple, une règle business complexe pourrait être la facturation d'un client. Cette règle peut être divisée pour obtenir les règles business élémentaires suivantes : calculer la taxe, obtenir les informations du client, produire la facture, etc. Si plusieurs règles business sont fort proches les unes des autres, la décision peut être prise de les rassembler pour en faire une seule et unique règle business. Après, il faut déterminer la valeur business de chaque règle élémentaire. Pour calculer cette valeur business, l'éditeur de logiciels doit se baser sur l'analyse du coût de développement (maintenance et remplacement) et la valeur commerciale annuelle prévue pour cette règle business. La valeur business est obtenue par la formule suivante : $\frac{Valeur\ commerciale - coût\ de\ maintenance}{coût\ de\ remplacement}$ [Sneed, 2006]. L'éditeur doit ensuite faire un choix et garder uniquement les règles business élémentaires qu'il juge profitable.

Arrivé à ce stade, l'éditeur de logiciels a déterminé les règles business élémentaires candidates. Ces règles élémentaires deviendront des fonctionnalités. Cependant, l'éditeur de logiciels peut décider de regrouper certaines règles élémentaires (car jugées inutiles quand celles-ci sont proposées individuellement en dehors du flux d'activités d'origine) pour en faire une règle business complexe, celle-ci deviendra alors une fonctionnalité. On notera qu'il est possible que l'application ait des ensembles de fonctionnalités prédéfinies (par exemple,

SalesForce.com offre son application CRM en 5 éditions différentes, la plus haute contenant toutes les fonctionnalités de l'ensemble). Dans ce cas, il n'est pas nécessaire d'effectuer une identification des règles business dans chaque édition, mais uniquement dans l'édition qui contient toutes les fonctionnalités de l'ensemble [Nassif et al., 2010].

Finalement, l'éditeur de logiciels se trouve en possession de la liste des fonctionnalités qui devront être externalisées. Il pourra donc passer à la prochaine étape qui consiste à concevoir les APIs.

4.4. Conception des APIS

L'étape de la conception des APIs peut être abordée une fois que les besoins sont identifiés, que le nouveau business model correspond aux attentes et est jugé viable et que l'éditeur de logiciels a déterminé les fonctionnalités qu'il souhaite externaliser. Il faudra effectuer des choix en matière de technologies tout en prenant en compte des critères tels que les standards à adopter, les contraintes actuelles, etc. Dans cette section, nous verrons les démarches et les méthodes qui existent dans la littérature sur le sujet. Nous passerons également en revue les tendances actuelles en ce qui concerne les technologies utilisées.

4.4.1. Approche pour la création d'une API

Aux entreprises qui souhaitent créer et exposer une nouvelle API, Eric Lundquist [Lundquist, 2012] conseille d'adopter une approche progressive. Cette approche débute par le développement de l'API pour une utilisation interne. Ensuite quand celle-ci est assez mature, l'entreprise peut la partager avec des partenaires commerciaux. Et finalement la rendre disponible à des tiers. Bien sûr, il faut ajouter à cela le volet marketing, une bonne gestion de l'écosystème API et une stratégie API. Concrètement, cela peut se faire en choisissant un business modèle et en élaborant une stratégie pour attirer les développeurs (fournir une bonne documentation, avec, si possible, des exemples concrets). L'éditeur de logiciels devra donc débuter par une APIzation dans un modèle fermé et puis, par la suite, étendre son modèle, s'il le souhaite, en adaptant et en modifiant les différents aspects pour obtenir un modèle ouvert restrictif et ensuite universel.

Cette dernière méthodologie peut être combinée avec une autre permettant de construire un « API management » efficace [Nassif et al., 2010; MuleSoft, 2015]. Cette autre méthodologie comprend des « best-practices » afin d'assurer que les APIs soient faciles d'utilisation, rapidement déployables, et qu'elles produisent le résultat métier attendu. Cette méthodologie comporte 7 étapes. La première d'entre elles, applique une approche de conception dite « API-First » (API en premier), c'est-à-dire qu'à la place de concevoir une application et ensuite construire une API par dessus, on crée d'abord l'interface et ensuite on implémente la logique métier derrière. Dans le cas d'une APIzation, il n'est pas possible de suivre cette approche puisque c'est à partir d'une application déjà existante que l'on souhaite concevoir des APIs. Comme nous l'avons précisé dans la section « approche adoptée », nous suivons une approche « bottom-up », on ne tiendra donc pas compte de cette première étape.

Voici, ci-après, les différentes étapes pour la phase de conception d'APIs lors du processus d'APIzation :

- L'éditeur de logiciels doit commencer par programmer les interfaces de programmation à proprement parler. Il les concevra dans le langage de programmation qui lui convient. On notera qu'il est préférable que la documentation soit rédigée avant l'implémentation de l'API car, dans le cas contraire, c'est le développeur/implémentateur qui rédige la documentation et il aura fréquemment tendance à écrire ce qu'il a fait. Cela conduit souvent à une documentation incomplète parce que le développeur est trop familier avec l'API et qu'il suppose que certaines choses sont évidentes alors qu'elles ne le sont pas [Henning, 2007].

- Après avoir conçu l'API il faut choisir le domaine d'exécution correct. Cette étape est cruciale, car elle aura des impacts sur le service, les responsabilités, la capacité à répondre aux attentes. Trois capacités clés auxquelles il est important de songer sont citées [MuleSoft, 2015] :

  - Le support hybride : il est important de développer les applications de manière à ce qu'elles puissent être migrées d'une machine locale vers le Cloud (et vice versa) sans devoir leurs apporter quelconques modifications. En effet, l'éditeur de logiciels pourrait à tout moment, par manque de ressources, décider de migrer une application sur le Cloud ou si cette application est déjà hébergée chez un fournisseur Cloud, de migrer l'application vers un autre fournisseur Cloud. Dans le cas d'une APIzation, l'application SaaS déjà existante devra peut-être être modifiée afin de prendre en compte un support hybride si cela est nécessaire.

  - Évolutivité, fiabilité, disponibilité : ces termes sont d'une importance capitale dans le succès d'une stratégie API. Il est donc essentiel de choisir des technologies permettant aux APIs d'atteindre ces différentes capacités.

  - Forte orchestration : c'est la capacité qu'une interface API et la logique métier implémentée derrière puissent correctement interagir, et cela malgré une complexité éventuelle de la partie métier.

- Une fois que la conception, le développement et l'exécution de l'API sur une plateforme fiable sont effectués, l'éditeur de logiciels pourra, pendant une période de test, utiliser l'API en interne afin de voir si elle atteint bien son but, si des modifications peuvent être effectuées pour l'améliorer, etc. Il obtiendra ensuite une version de son API pouvant être utilisée en production.

À ce stade, l'éditeur de logiciels a procédé à une APIzation dans un modèle fermé. Les étapes suivantes sont surtout importantes pour les modèles ouverts restrictif et universel.

- Une fois que l'éditeur de logiciels juge que l'API est assez mature, il va la partager avec un ou plusieurs partenaires. Pour cela, il faut pouvoir exposer l'API pour la rendre accessible aux partenaires qui en ont besoin. Pour qu'une API puisse être découverte et accessible, un dépôt de service central est utilisé. Ce dernier facilite également la catégorisation et la recherche à travers les services en offrant une vue consolidée des APIs. La visibilité peut

être utilisée comme un indicateur clé d'une API afin de prendre les mesures correctives adéquates si elle n'est pas correctement atteinte. Nous verrons plus en détail dans une prochaine section comment l'éditeur de logiciels peut exposer ses APIs.

- Il devra ensuite gérer les services via les versions, règles et contrats. Grâce aux versions, le fournisseur peut savoir qui utilise ses APIs, quelles versions les consommateurs utilisent et comment ils les utilisent. Cela lui permet de mieux gérer le cycle de vie de ses APIs et de pouvoir évaluer l'impact d'un retrait de celles-ci. Les règles et contrats permettent d'augmenter la sécurité ainsi que de gérer les SLAs (Service-Level Agreement, document qui définit la qualité de service requise). L'éditeur de logiciels devrait concevoir une solution API de manière à permettre la création de règles et de contrats bien définis et pouvoir les associer correctement avec les APIs et les consommateurs.

- Promouvoir et socialiser les APIs : le succès d'une API est très lié à la communauté de développeurs construite autour d'elle. Afin de promouvoir au mieux l'API, il est important de faciliter aux utilisateurs la consommation de celle-ci, notamment en facilitant son suivi, en mettant à leur disposition de la documentation téléchargeable, et en répondant à leurs questions. Ce point est tout particulièrement important dans le cadre d'une APIzation dans un modèle ouvert universel.

- Surveiller et évaluer l'utilisation de l'API : l'éditeur de logiciels peut surveiller, sur une période de temps définie, l'utilisation de ses APIs du point de vue technique et commercial. Cela lui permettra de mieux comprendre les consommateurs et utilisateurs, et en fin de compte, de pouvoir proposer de meilleurs services.

- Amélioration continuelle : afin d'améliorer l'expérience utilisateur et la productivité, il est très conseillé d'optimiser les APIs. La meilleure façon de le faire est de réadapter l'API en effectuant plusieurs fois les étapes que nous venons juste d'aborder.

Ces différentes étapes font toutes parties du cycle de vie de l'API (figure 18).

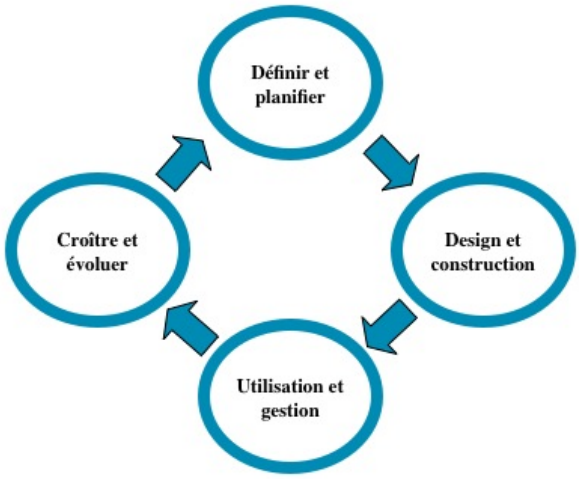

Figure 18 – Cycle de vie d'une API.

4.4.2. APIs accessibles à distance : les Web Services

Afin que les APIs puissent être utilisées par les consommateurs, il est nécessaire qu'elles puissent être rendues accessibles à travers le réseau. Dans ce cas, on parlera de Web Services. Pour créer des Web Services, l'éditeur de logiciels devra aborder les différentes couches (voir point 2.1.7.2), faire un choix au niveau des technologies et standards à utiliser. Dans cette sous-section nous aborderons les tendances actuelles en terme de technologies utilisées pour la création de Web Services. L'aspect sécurité sera également pris en compte.

4.4.2.1. Tendances technologiques actuelles

En 2014, le protocole le plus utilisé est REST (figure 19). Quant aux formats de données, c'est l'XML qui arrive en tête suivi de JSON (figure 20). Sur les 82% des APIs qui utilisent XML ou JSON, un nombre assez important supporte les deux formats à la fois (figure 21). [Source : ProgrammableWeb, repris dans Gat et al., 2013]

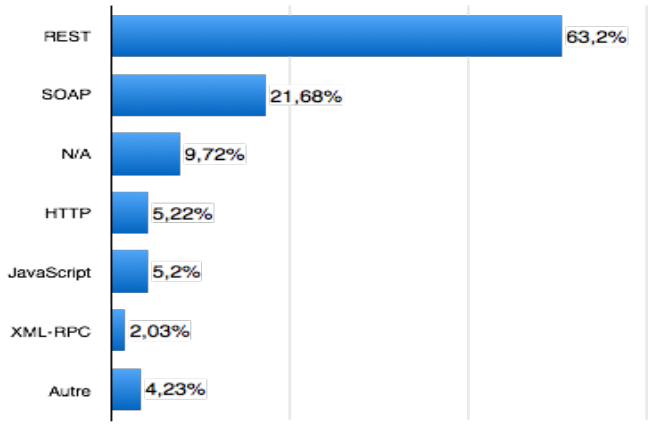

Figure 19 – Pourcentage d'APIs supportant un protocole spécifique.

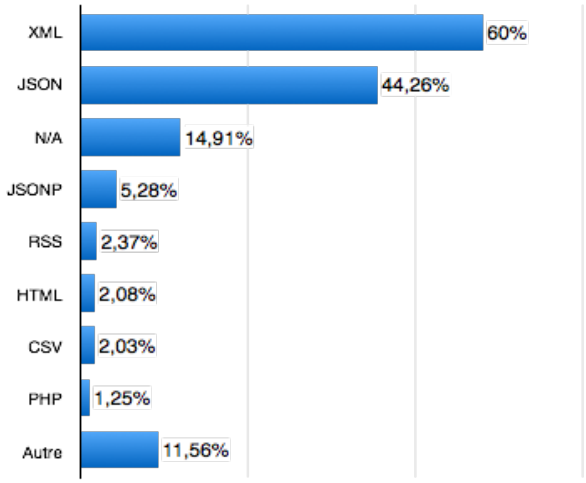

Figure 20 – Pourcentage d'APIs supportant des formats de données spécifiques.

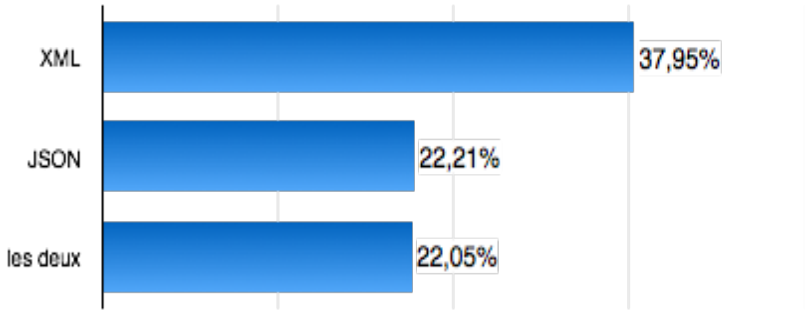

Figure 21 - Pourcentage d'APIs supportant JSON ou XML ou les deux.

Du point de vue de la sécurité et de la confidentialité, il est recommandé d'utiliser les bonnes pratiques connues dans le domaine des technologies web telles qu'utiliser le protocole HTTPS quand cela est possible pour éviter la transmission de données importantes en texte clair, pour contrôler les inputs, éviter des attaques par injection (validation du contenu du message), contrer les attaques par déni de service (DoS), etc.

Certains utilisent le protocole d'autorisation web OAuth 1.0 ou 2.0 (et d'autres solutions basées dessus) comme solution au défi que sont l'authentification et la gestion d'identité. Il existe deux solutions construites par des communautés standards : OpenID Connect pour l'authentification et User-Managed Access (UMA) pour la gestion d'accès [Gat et al., 2013 ; Kanmani et al., 2013].

La plupart du temps, l'utilisation de REST sera suffisante, mais, dans certains cas, il vaut mieux utiliser SOAP. Les différents cas qui pourraient pousser les développeurs à adopter SOAP plutôt que REST, sont les suivants [Castillo et al., 2011 ; Rozlog, 2010 ; Spies, 2008]:

- Utilisation d'un autre protocole que HTTP/HTTPS : REST est lié au protocole HTTP/HTTPS contrairement à SOAP qui peut utiliser presque n'importe quelle couche de transport (SMTP, JMS, etc.)

- Garantir un niveau de fiabilité et de sécurité : SOAP offre des normes supplémentaires pour les assurer. Un exemple est la spécification WSRM pour la fiabilité.

- Utilisation de contrats formels : si le fournisseur et le consommateur doivent se mettre d'accord sur le format d'échange des messages, SOAP est conseillé, car il fournit un cadre strict pour ce genre d'interaction.

- Opérations avec état : REST étant « sans état » (state-less), chaque requête doit contenir toutes les informations requises au traitement. Dans le cas où il est nécessaire d'utiliser des informations contextuelles et de gérer l'état conversationnel, SOAP se révèle plus adéquat, car il est en mesure de soutenir ces besoins grâce aux spécifications dans la structure WS*.

4.5. Livraison des APIS

Une fois les Web Services créés, il faut adopter une méthode de livraison pour les amener aux consommateurs. Dans les points suivants, nous commencerons par expliquer quelles sont les méthodes existantes pour y procéder. Ensuite nous envisagerons les raisons qui mènent à penser que l'architecture SOA constituerait une bonne solution pour livrer des APIs. Nous clarifierons les différences entre un SaaS et l'architecture SOA. Finalement, nous proposerons la suite de la méthodologie pour fournir des Web Services en utilisant SOA.

4.5.1. Les méthodes de livraison existantes

Le fournisseur d'APIs dispose de plusieurs moyens pour livrer ses APIs aux consommateurs. Un premier moyen est de fournir les informations nécessaires (par exemple l'URI) pour avoir accès au Web Service. Dans ce cas, l'API sera hébergée dans l'infrastructure du fournisseur et celui-ci devra donc configurer son système afin qu'une requête extérieure puisse être acheminée jusqu'à l'API. De plus, il devra lui-même mettre en place les différents outils permettant de surveiller l'utilisation et les performances de ses APIs. L'entreprise qui choisira cette méthode de livraison pourra gérer de A à Z toute l'infrastructure API et choisir les outils et technologies qu'elle souhaite utiliser. Ce choix entraine toutefois un travail conséquent et implique d'être en possession des différentes ressources nécessaires. Ce moyen est surtout utilisé dans le cadre d'un modèle fermé ou ouvert restrictif.

Pour les fournisseurs qui ne souhaitent pas gérer eux même l'infrastructure API et qui souhaitent passer par un courtier/intermédiaire pour distribuer ses APIs (surtout dans le cadre d'un modèle ouvert universel), il existe de nombreuses sociétés qui mettent à leur disposition des plateformes de gestion d'APIs [Raivio et al., 2011]. Dans le cas d'une utilisation de la plateforme tout Cloud, le fournisseur mettra son API sur la plateforme de gestion d'APIs, la liaison de celle-ci avec le système back-office de l'entreprise s'effectuant via une passerelle sécurisée. Les consommateurs pourront accéder à l'API via la plateforme en ligne. Notre analyse de différentes solutions de gestion d'APIs offertes sur le web (par exemple 3Scale, Apigee, WSO2) nous permet de constater qu'elles incluent généralement plusieurs parties :

- Un portail de développeurs : ce portail permettra aux développeurs tiers de trouver l'API, de comprendre son utilisation et de s'enregistrer afin d'obtenir un accès pour utiliser l'API.
- Une passerelle API : cet élément permet de sécuriser et d'assurer la médiation du trafic entre les APIs et les systèmes back-office (où se trouve toute la logique métier).
- Le gestionnaire du cycle de vie API : il permet de gérer les processus de conception, de développement, de déploiement, de versioning et le retrait d'API.
- Le gestionnaire de contrats API : ce gestionnaire permet de gérer les accès à l'API, de définir les accords sur les niveaux de services (SLA) et de surveiller en temps réel si l'API respecte bien les termes du contrat.
- Le gestionnaire de politique API : le fournisseur de l'API pourra gérer les règles en matière de sécurité, d'étranglement, de limitation de débit, et la restriction en matière de comportement. Ces modifications peuvent être effectuées sans devoir arrêter l'API.

- L'outil de surveillance de l'API : cet outil va permettre la surveillance de l'utilisation de l'API ainsi que des performances. La majorité des solutions permettent d'analyser la consommation sous différents points de vue, par exemple, par API, par version de l'API, par (type de) consommateur, etc.

Ces solutions peuvent être déployées de trois manières différentes :

- Sur site : l'entreprise déploiera la solution sur sa propre infrastructure. Elle pourra ainsi avoir un contrôle total et gérer elle même la conformité aux normes. Elle établira ses propres règles.
- Sur API Plateforme-as-a-Service : la solution sera hébergée sur le Cloud, et non chez le concepteur de l'API. Cette solution est particulièrement intéressante pour les entreprises n'ayant pas les ressources, les connaissances nécessaires ou ne souhaitant pas gérer la plateforme.
- Hybride : dans ce cas, seulement une partie sera déployée dans l'infrastructure de l'entreprise (par exemple le gestionnaire API) tandis que le reste sera hébergé dans le Cloud (il s'agira souvent de la partie permettant de construire et d'élargir la communauté de développeurs).

Comme nous l'avons expliqué dans le premier paragraphe de cette section, un fournisseur peut créer lui-même sa plateforme d'APIs Management (figure 22).

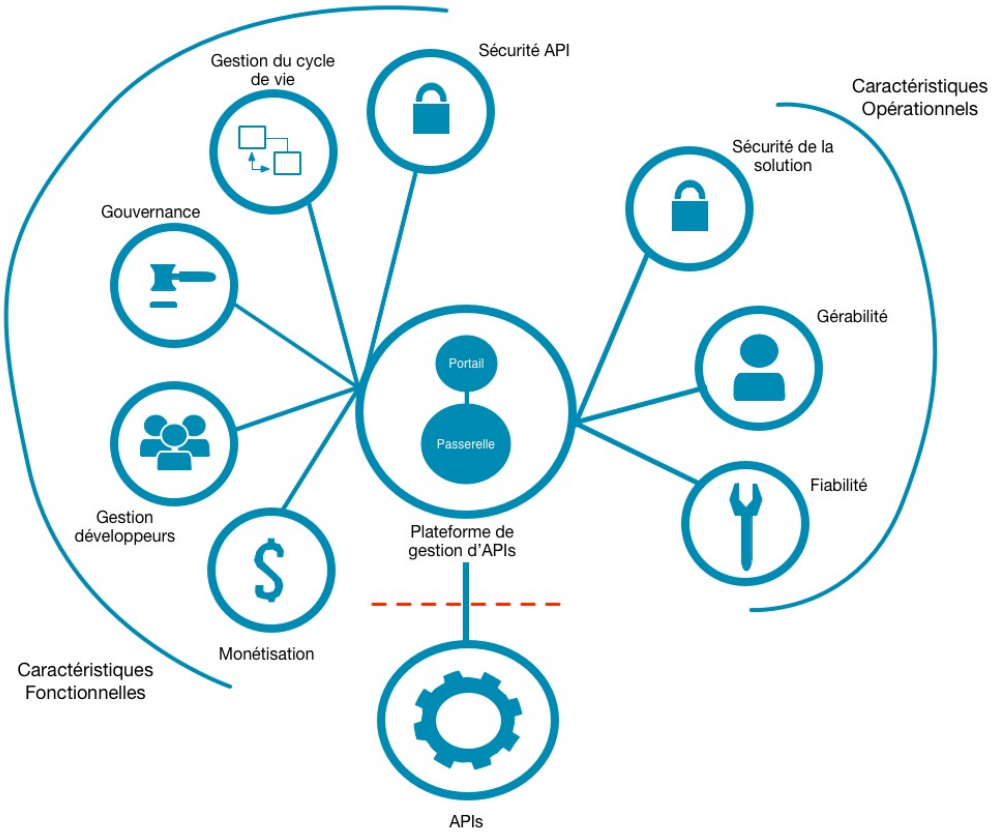

Figure 22 – Plateforme d'APIs Management [inspiré et adapté de See Technologies, 2015; Vordel, 2015]

Pour cela, il devra définir une architecture adaptée aux APIs. Dans les sections suivantes, nous aborderons l'architecture qui semble la mieux adaptée et nous préciserons une méthode à suivre pour la mettre en place.

4.5.2. Pourquoi l'architecture SOA est-elle une bonne candidate ?

Comme nous l'avons vu dans la partie 2 « cadre théorique », SOA correspond à un style architectural qui permet de maintenir une certaine interopérabilité entre les composants, ce qui facilite grandement l'utilisation des services dans des infrastructures différentes. De plus, SOA et les APIs partagent les mêmes piliers [MuleSoft, 2015] :

- Les services doivent être réutilisables : les APIs peuvent être définies comme publiques de manière à ce qu'elles puissent être réutilisées par d'autres applications.
- Les services doivent avoir un contrat : il est tout à fait possible de définir un contrat spécifique pour chaque API. Un contrat rassemble un ensemble de règles devant être suivies.
- Les services peuvent être facilement découverts et utilisés: la grande majorité des APIs sont fournies avec une documentation et des exemples concrets.
- Les services peuvent être composés à partir d'autres services : une API peut être composée à partir d'autres APIs (par exemple l'API Analytics de Klaviyo est composée de l'API People et l'API Events).

Dans une architecture SOA, la partie qui englobe la gestion des services, leur accès et la définition des SLAs, est appelée SOA governance (gouvernance SOA en français). Avec les APIs, c'est comparable, mais à plus grande échelle. Comme décrit dans les précédentes sections, il faut gérer les différents types d'APIs et de consommateurs (qui peuvent être des partenaires ou développeurs tiers), c'est ce qu'on appelle l'API management (gestion des APIs en français). Ce concept est très similaire au « SOA governance ».

4.5.3. Distinction SaaS et SOA

Étant donné qu'il existe souvent une confusion entre les termes SaaS et SOA, certains pourraient mettre en doute l'intérêt d'effectuer une démarche d'APIzation à partir d'une application SaaS car on obtiendrait alors un résultat similaire à la situation d'origine. Il est donc préférable de clarifier cette confusion en effectuant une distinction entre ces deux termes sur base de plusieurs articles trouvés dans la littérature [Laplante et al., 2008; Zhang et al., 2007; Turner et al., 2003].

SaaS est un modèle de livraison de programme informatique tandis que SOA est un style architectural dont les éléments qui constituent le système informatique sont des services réutilisables. Le modèle de Zachman [Zachman, 1999] peut être utilisé afin de bien distinguer les deux termes (figure 23). Il permet de représenter les différents points de vue des parties prenantes pour un modèle architectural.

Perspective parties prenantes	Données	Fonction	Réseau
Objectif/Scope	Liste des entités importantes pour le métier	Liste des processus que le métier effectue	Emplacement dans lequel le métier fonctionne
Modèle d'affaires	Représentation des entités business et des règles	Représentation des ressources business et des processus	Représentation logique des unités métier
Modèle du système d'information	Spécifications des exigences des données et des objets	Spéciations des exigences pour l'interaction entre les données et les objets	Architecture logiciel ou système
Modèle technologique	Spécifications de la conception des données et objets	Spécification de la conception pour les interactions entre les données et objets	Composants logiciels et matériels
Représentation détaillée	Description de base de données	Code	Architecture réseau
Fonctionnement du système	Données et objets	Fonction ou interaction	Communications

Tableau 2 – L'ensemble des modèles architecturaux de Zachman sur base des points de vue des différentes parties prenantes [Laplante et al., 2008]

Explication :
- Objectif/Scope est l'ensemble des objectifs et cadres qui donne une vue approximative du système informatique (ex : en se basant sur des use cases)
- Business model correspond à la représentation du propriétaire
- Modèle du système d'information correspond à la représentation du designer
- Modèle technologique correspond à la représentation du constructeur du système
- Représentation détaillée correspond à la représentation du système hors contexte
- Fonctionnement du système correspond à la représentation du système lui-même

Ce modèle de Zachman peut être adapté pour comparer SaaS et SOA. En effet nous pouvons nous focaliser uniquement sur l'aspect réseau (colonne « réseau»). Le tableau suivant est alors obtenu [Laplante et al., 2008] (figure 24) :

Perspective parties prenantes	Réseau (SOA)	Réseau (SaaS)
Objectif/Scope	Liste des services possibles à utiliser	Liste des services possibles à délivrer
Modèle d'affaires	Liste des services business à utiliser	Liste des services business à fournir

Modèle du système d'information	Modèle d'interaction des composants du service	Modèle d'interaction des composants
Modèle technologique	Modèle d'interaction des composants du service dépendant de la technologie et de la plateforme	Modèle d'interaction des composants dépendant de la technologie et de la plateforme
Représentation détaillée	Liste des technologies dépendantes des langages et des protocoles utilisés (tel que UDDI, SOAP, XML, WSDL) et des services actuellement utilisés	Architecture publication-souscription et mise en place de notifications ; liste des technologies dépendantes du langage, protocoles et services utilisés (s'il y en a)
Fonctionnement du système	Communication interservices, coordination et collaboration	Communication intercomposants, coordination et collaboration

Tableau 3 – Modèle de Zachman adapté pour une comparaison de SOA et SaaS [Laplante et al., 2008]

Plusieurs différences sont notables. Premièrement, du point de vue des objectifs et du business model, SOA se focalise sur l'aspect utilisation des services, tandis que SaaS se focalise sur l'aspect livraison des services.
Deuxièmement, du point de vue du designer, SOA est un modèle architectural décrivant un schéma d'interaction des composants de service le constituant tandis que SaaS décrit un schéma d'interaction des composants le constituant, mais qui ne sont pas forcément des services.
Ensuite du point de vue constructeur, SOA et SaaS doivent identifier et utiliser une technologie (par exemple service web) pour mettre en place les modèles d'interactions du système informatique.
Malgré quelques différences significatives, lorsque nous en parlons dans un contexte de systèmes d'informations à grande échelle, ces deux modèles architecturaux sont étroitement liés et peuvent être combinés. Lors du processus d'APIzation d'une application (SaaS ou autre) la combinaison de ces deux modèles est donc une solution. Dans la sous-section suivante, nous aborderons la manière pour utiliser SOA dans le cadre d'une APIzation.

4.5.4. Convergence SaaS et SOA

4.5.4.1. But recherché

Il existe une forte tendance qui consiste à vouloir concevoir des logiciels informatiques comme un service utilisable à distance par toute une série d'utilisateurs. SaaS est tout indiqué pour cela. Dans le but d'effectuer une APIzation de ces logiciels SaaS pour rendre disponibles des fonctionnalités précises d'un programme sous forme de services spécifiques à des utilisateurs ou systèmes externes, une solution serait donc de combiner SaaS avec SOA afin de définir une architecture d'ensemble de services qui peuvent être distribués facilement.

4.5.4.2. Problèmes rencontrés avec SOA

Ces dernières années, la majorité des organisations utilisent l'architecture SOA pour rationaliser les processus, augmenter l'efficacité et réduire les coûts. Elles y arrivent en ayant de bons résultats dans un premier temps. Mais ces organisations adoptent une mauvaise approche lors de la conception de leur projet SOA [MuleSoft, 2015]. En effet, elles suivent une approche dite « Top-Down », ce qui a pour effet de créer de nombreux problèmes au sein de l'organisation et conduit à une augmentation du coût de manière potentiellement élevée. Leur approche utilise la stratégie « Rip-and-replace » (jeter et remplacer) qui a pour but de remplacer les systèmes vieillissants d'une entreprise par des systèmes modernes. Le problème est que ces projets prennent souvent plus de temps que prévu, leur temps de réalisation peut déborder sur plusieurs années, ce qui a pour conséquence finale l'augmentation de la consommation des ressources et le dépassement de budget planifié.

Alors que beaucoup ont critiqué et rejeté l'utilisation de cette architecture en raison des problèmes engendrés, les principes fondamentaux de SOA sont actuellement très utiles et permettent une nouvelle gestion des APIs.

4.5.4.3. Solution retenue

Une solution consiste en l' « API management » [MuleSoft, 2015; Nassif et al., 2010] (Gestion des APIs). Cette solution reprend de nombreux principes de l'architecture SOA, telles que l'abstraction de la logique métier, l'évolutivité, la réutilisation des services, la flexibilité, l'interopérabilité entre les services et applications. Mais elle contient deux différences importantes permettant ainsi d'éviter les problèmes liés à l'utilisation de SOA [MuleSoft, 2015].

La première différence consiste en l'utilisation de la stratégie « Bottom-Up », wrap-and-renew en lieu et place de la stratégie de « Top-Down », rip-and-replace. Ce qui se traduit par l'ajout d'interfaces accessibles au dessus des fonctionnalités. Ces interfaces permettent de protéger les différents utilisateurs des technologies sous-jacentes utilisées lors de l'implémentation de la logique métier. L'utilisation de cette stratégie apporte un gain au niveau de l'agilité business [Kana, 2013]. Cela est donc compatible avec notre approche adoptée pour effectuer une APIzation.

La deuxième différence est que les APIs sont directement liées au ROI (retour sur investissement). Les projets comprendront des étapes importantes (jalons) lors desquelles des artefacts du programme pourront rapidement être implémentés et utilisés. Cela facilite la mesure du retour sur investissement.

Cette solution apporte plusieurs bienfaits, tels que l'accélération du développement de l'application, la réduction du risque d'échec et l'amélioration de l'efficacité des développeurs.

4.5.5. Association des Web Services et SOA

À ce stade l'éditeur de logiciels est en possession de Web Services. Ceux-ci sont dès lors accessibles via le réseau, mais aucune garantie n'est faite quant à leur indépendance et leur

généricité. Cela dépendra du protocole utilisé au niveau de la couche « message » de la pile des interactions du service (REST, SOAP).
Sur base de la solution retenue au point suivant, l'éditeur de logiciels pourra les exposer de manière efficace aux consommateurs en utilisant le style architectural SOA. Pour cela il devra commencer par une vérifier la conformité des Web Services construits. À cette fin, on se basera sur leur potentiel de réutilisation, leur indépendance et leur caractère générique (par exemple, la technologie REST ne permet pas de concevoir des services aussi indépendants que SOAP) [Nassif et al., 2010].
Finalement, l'éditeur de logiciels est prêt pour fournir ses Web Services via un courtier de service (service broker).

Pour résumer la méthodologie proposée, la figure 25 illustre les étapes clés du processus d'APIzation. [Nassif et al., 2010]

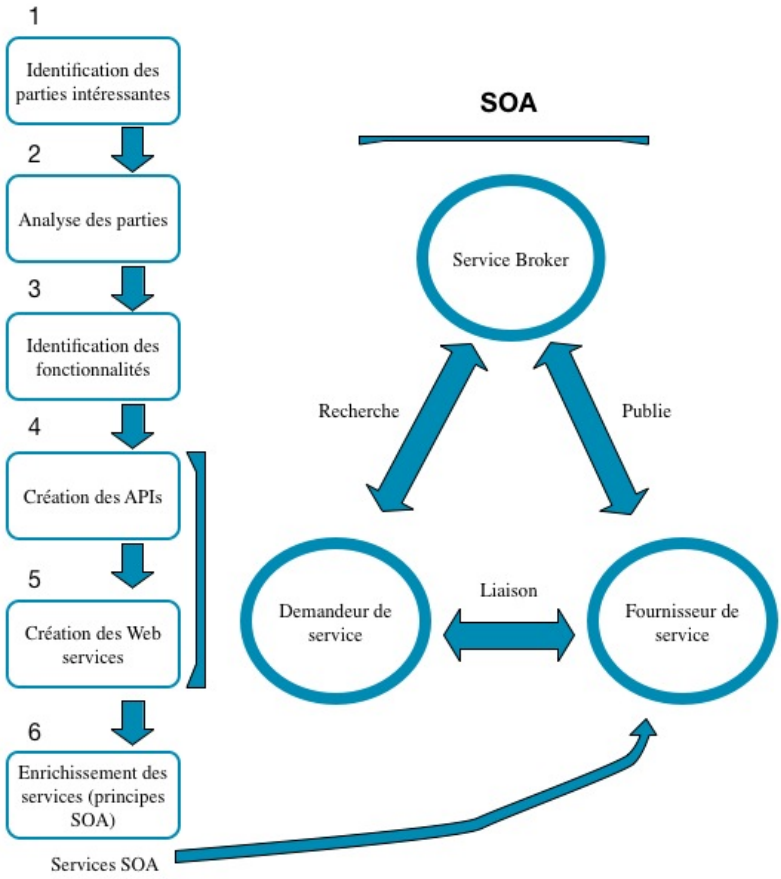

Figure 23 – Externalisation des fonctionnalités d'une application en services et distribution de ceux-ci en suivant le style SOA [inspiré et adapté de Nassif et al., 2010]

5

CONCLUSION

L'objet de ce mémoire est d'apporter un éclairage sur le processus décrit sous le terme « APIzation » et de proposer une méthodologie pour le développer. Dans la deuxième section de notre travail, nous avons été amenés à introduire les concepts de base nécessaires à la compréhension du domaine auquel ce terme est lié. Ainsi nous avons défini et expliqué ce qu'est le Cloud Computing. Nous avons vu que le Cloud Computing possède cinq caractéristiques essentielles : le self-service, l'accès par le réseau, la mise en commun des ressources, l'élasticité rapide et le service de monitoring. Il existe quatre modèles de déploiement : le cloud privé, le cloud communautaire, le cloud public et le cloud hybride. Enfin il existe trois modèles de services pour le Cloud Computing : SaaS, PaaS et IaaS. Ensuite de quoi, nous avons décrit le style architectural SOA qui permet de mettre en place un mécanisme simple et efficace pour donner la possibilité aux fournisseurs de mettre à disposition des consommateurs leurs services via le réseau. Après, nous avons abordé les Web Services avec leur fonctionnement et leur structure ainsi que les APIs. Nous avons identifié trois types d'APIs (privé, public et partenaire), chacun étant utilisé dans des situations bien précises (API privée : en interne, API publique : pour tout le monde, API partenaire : pour les partenaires commerciaux). Une définition de l'interopérabilité, caractéristique essentielle dans ce domaine, a ensuite été apportée et discutée.

Enfin, pour terminer cette deuxième section, nous avons proposé une définition du terme APIzation. Une APIzation est le fait d'externaliser des fonctionnalités d'une application existante en créant un ensemble d'APIs. Par "externaliser", nous entendons l'introduction d'une interface au-dessus d'une fonctionnalité. Cette interface va ainsi permettre aux utilisateurs d'accéder à une fonctionnalité bien spécifique. Nous avons identifié plusieurs modèles d'APIs : 1) le modèle fermé qui est destiné à une utilisation interne des APIs 2) le modèle ouvert qui vise à exposer les APIs à des personnes extérieures de l'entreprise. Deux sous-modèles ouverts existent : le sous-modèle restrictif qui sera utilisé pour restreindre et réserver l'accès des APIs aux partenaires de l'entreprise et le sous-modèle universel qui, comme son nom l'indique, permet à tous les utilisateurs d'utiliser les APIs de la même manière.

Dans la troisième section de notre travail, nous avons abordé l'aspect économique. Pour ce faire, nous avons débuté par définir le concept d'API Economy auquel toute entreprise devra se préparer lorsqu'elle décide d'effectuer une APIzation. Différents acteurs y ont été identifiés (le fournisseur, le consommateur, et l'utilisateur final) ainsi que les avantages et opportunités qu'elle offre. Nous avons ensuite traité des business models (modèles d'affaire) en définissant ce qu'est un business model d'une manière générale et en présentant ensuite les différents

modèles de revenus APIs existants. Quatre grandes catégories ont été identifiées dans la littérature: gratuit, le consommateur paie, le consommateur est payé, indirect. De plus, nous avons distingué quatre classes d'APIs qui permettent à une entreprise de s'orienter dans son choix de modèle de revenus : l'API est le produit, l'API projette le produit, l'API favorise le produit, l'API nourrit le produit. Pour conclure cette section, nous avons pointé les changements apportés, du point de vue de l'activité commerciale, par cette nouvelle économie dans l'entreprise en utilisant le Business Model Canvas d'Osterwalder. Plusieurs changements majeurs ou mineurs ont pu être identifiés.

Dans la quatrième et dernière section, nous avons passé en revue les divers défis (et la manière de les surmonter) auxquels un éditeur de logiciels devra souvent faire face lorsqu'il se lance pour la première fois dans une démarche d'APIzation. Nous avons ensuite proposé une méthodologie à suivre afin d'effectuer au mieux une APIzation. Nous avons commencé par analyser les approches ascendante et descendante pour au final adopter l'approche la plus adéquate pour le processus d'APIzation : l'approche ascendante. Ensuite, étant donné qu'il n'est pas toujours aisé pour un éditeur de logiciels de déterminer correctement les fonctionnalités qui doivent être externalisées, une approche en quatre étapes est proposée : déterminer les parties de l'application intéressantes et utiles, étudier et analyser ces parties pour identifier leurs spécifications afin de pouvoir déterminer le niveau de dépendance entre les fonctionnalités, découper ces fonctionnalités en règles business élémentaires et évaluer leur valeur commerciale et, enfin, sur base des règles business retenues, reconstruire les fonctionnalités finales qui devront être externalisées. En ce qui concerne la conception des APIs, une approche en huit étapes est proposée sur base de plusieurs articles trouvés dans la littérature. Cette approche définit les étapes à effectuer dans les différentes phases du cycle de vie de l'API. Nous avons également repris quelques statistiques sur les technologies actuellement utilisées pour créer des APIs et les distribuer via le réseau. Finalement, nous avons abordé l'étape de livraison des APIs. Un éditeur de logiciels aura le choix dans les méthodes de livraison. Il pourra faire appel à une société externe pour utiliser une plateforme de gestion d'APIs en ligne. Mais il aura également la possibilité de gérer cette partie lui même. Pour cela, nous avons d'abord précisé les différences entre un SaaS et le style architectural SOA. Différentes approches de mise en place du style architectural SOA ont été analysées et comparées pour, au final, confirmer la continuité d'une utilisation de l'approche ascendante. Nous avons terminé cette section en expliquant les dernières étapes pour distribuer les Web Services avec le style architectural SOA.

À l'avenir, en vue d'une amélioration et d'un affinement de la méthode proposée, il serait intéressant d'effectuer des interviews d'éditeurs de logiciels afin de valider la complétude d'une telle méthode et apporter des précisions complémentaires relatives aux technologies utilisées. Il serait également fort utile de relever les aspects jugés les plus importants pour un éditeur dans ce domaine ainsi que le pourcentage respectif d'adoption de chaque modèle d'APIs (fermé, ouvert restrictif/universel).

ANNEXES

6.1. Liste non exhaustive de plateformes de gestion APIs

Nom	URI
3Scale API Management plateforme	http://www.3scale.net/api-management/
Akana API Management	https://akana.com/solution/api-management
Apiary	https://apiary.io/products
ApiAxle	http://apiaxle.com
Apigee Edge	http://apigee.com/about/products/apis-and-edge
Axway 5 Suite	https://www.axway.com/fr/solutions-d-entreprise/gestion-des-API
CA API Management	http://www.ca.com/us/products/api-management.aspx?intcmp=headernav
IBM API Management on Cloud	http://www-03.ibm.com/software/products/en/api-management-cloud
Mashape	https://www.mashape.com
Mashery API Management	http://www.mashery.com/api-management/saas
MuleSoft API Manager	https://www.mulesoft.com/platform/api/manager
Oracle API Management	http://www.oracle.com/us/products/middleware/soa/api-management/overview/index.html
SAP API Management	http://www.sap.com/pc/tech/business-process-management/software/api-management/index.html
See Technologies	http://seetechnologies.co/api-management-development.php
WSO2 API Management	http://wso2.com/api-management/

6.2. Description du business model canvas

6.2.1. Le segment clientèle

Toute entreprise a besoin d'une clientèle (rentable) afin de maintenir son activité sur le long terme. Il est important de bien cerner les différents segments clientèle afin de comprendre ses besoins, ses comportements, de pouvoir justifier la raison pour laquelle certains segments sont ignorés. Il existe différents types de segments clientèle sur lesquels une entreprise pourrait se focaliser (marché de masse, marché de niche, segmenté, diversifié).

6.2.2. La proposition de valeur

Dans ce bloc, sera décrite la proposition de valeur, c'est-à-dire le produit ou le service qui permettra d'apporter de la valeur pour un segment clientèle spécifique. Cette valeur ajoutée répondra à un problème ou satisfera des besoins d'un client. S'il existe des concurrents sur le marché, l'entreprise veillera à se différencier de ceux-ci avec des fonctionnalités ou des

attributs ajoutés. On notera que l'offre proposée aux clients est n'est pas uniquement constituée d'un produit ou d'un service, elle peut comporter d'autres éléments qui apporteront une valeur ajoutée à l'offre (par exemple, l'accessibilité, la réduction des risques, la performance) [36].

6.2.3. Les canaux

L'entreprise décrira ici les différents canaux qu'elle compte utiliser pour interagir avec les clients. Ces canaux peuvent être utilisés pour atteindre différents buts : la communication, la distribution, la vente, etc. Étant donné que les canaux constitueront les points de contact avec les clients, l'entreprise pourra s'efforcer de maintenir un certain niveau d'expérience client. Ces canaux servent plusieurs fonctions [36] : sensibiliser les clients à l'offre de l'entreprise, aider les clients pour qu'ils puissent évaluer la proposition de valeur de l'entreprise, permettre la vente de produits et services spécifiques aux clients, livrer la proposition de valeur aux clients et fournir un service après-vente.

6.2.4. Les relations clients

Une entreprise doit spécifier et décrire le type de relation qu'elle souhaite établir entre elle et les segments clientèle spécifiques. La relation peut être effectuée via un contact personnel ou aller jusqu'à l'utilisation d'un processus automatique. Afin d'établir les relations clientèle, l'entreprise doit se baser sur les motivations suivantes : acquisition de nouveaux clients, fidélisation du client, stimulation des ventes. Elle choisira le type de relation en fonction du segment clientèle cible et de la fonction de cette relation.

6.2.5. Les flux de revenus

Les flux de revenus vont représenter l'argent que l'entreprise va générer suite à la vente de sa proposition de valeur aux clients. Pour chaque segment clientèle, l'entreprise pourra mettre en place un ou plusieurs types de flux de revenus qui se différencieront en fonction du mécanisme de tarification. Il en existe deux types : les revenus ponctuels et les revenus récurrents. Ensuite l'entreprise choisira le moyen de générer un revenu (vente d'actif, frais d'utilisation, frais d'abonnement, location, vente de licence, frais de courtages, publicités). Le choix du mécanisme de tarification est crucial, car il peut avoir un impact très significatif en terme de revenu généré. De plus, il est possible d'adopter un mécanisme de tarification fixe (avec des prix prédéfinis basés sur des variables statiques) ou une tarification dynamique (avec des prix qui changent en fonction des conditions du marché).

6.2.6. Les Ressources clés

Dans ce sixième bloc, seront décrites les ressources nécessaires afin que le business model de l'entreprise puisse fonctionner. Ces ressources seront différentes en fonction du business model, mais dans tous les cas, elles doivent permettre à l'entreprise de créer et offrir une valeur de proposition, atteindre des marchés, maintenir la relation avec les segments clientèle et gagner un revenu. Elles peuvent être de nature physique (immobilier, véhicules, machines, systèmes…), financière (trésorerie, crédits, stock options), intellectuelle (marques, connaissances propriétaire, brevets et copyright, partenariat) ou humaine. L'entreprise pourra

acquérir ces ressources en les achetant ou en les louant, mais également par l'intermédiaire des partenaires clés.

Par exemple, Microsoft et SAP dépendent de leurs logiciels et de leurs propriétés intellectuelles qu'ils ont développés tout au long des années.

6.2.7. Les activités clés

Dans cette partie, sont décrites les activités qui seront indispensable au bon fonctionnement du business model. Ces activités constituent les actions les plus importantes au sein de l'entreprise. Tout comme les ressources clés, les activités clés dépendent du type de business model et elles sont nécessaires pour la création et l'offre de la valeur de proposition, l'atteinte des marchés, le maintien de la relation avec les clients. Elles peuvent faire partie de l'une des catégories suivantes : production (qui comprend les activités liées à la production d'un produit), résolution de problème (qui comprend les activités liées à la résolution de problèmes telles que la gestion des connaissances ou la formation continue), plateforme/réseau (qui comprend les activités liées à la gestion de la plateforme. Par exemple, le model business d'eBay nécessite le développement et le maintien de sa plateforme).

6.2.8. Les partenaires clés

Afin de renforcer leur business model, les entreprises peuvent créer des partenariats avec d'autres entreprises. Elles pourront ainsi, par exemple, réduire les risques, les coûts, acquérir de nouvelles ressources, profiter d'une expertise dans un domaine particulier, etc. En fonction de ce qu'elle recherche, une entreprise pourra choisir parmi quatre types de partenariats : une alliance stratégique entre entreprises non concurrentes, coopétition (partenariats entre concurrents, par exemple Apple et Samsung sont concurrents dans le domaine de la téléphonie, mais Samsung est un fournisseur de composants matériels pour les Smartphones d'Apple), relation acheteur-fournisseur pour assurer une fiabilité de l'approvisionnement, coentreprise afin de développer un nouveau business.

6.2.9. La structure des coûts

Pour terminer, ce dernier bloc comprendra une description des coûts les plus importants afin que le modèle puisse tenir. Cela permettra à l'entreprise de déterminer les bénéfices qu'elle pourrait retirer de son activité. Certaines entreprises spécialisées dans le low-cost apporteront une très grande importance à ce bloc en essayant de minimiser au maximum les coûts encourus par leurs activités.

LISTE DES FIGURES

LISTE DES TABLEAUX

BIBLIOGRAPHIE

Par ordre alphabétique

Ajami D, The API Economy - Grasp the rocket or get left behind, http://www.absi.be/api-economy.aspx, 1 Juin 2015, (Accès le : 12 Août 2015).

Alagarasan V., "API Management Introduction and Principles", in *Service Technology Magazine*, Arcitura Education, pages 4-11, Mars 2015.

Almonaies A. A., Alafi M. H., Cordy J. R., Dean T. R., "Towards a framework for migrating web applications to web services", *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pages 229-241, 2011.

Alonso G., Casati F., Kuno H., Machiraju V., "Web Services", in Web Services - Concepts, Architectures and Applications, Springer-Verlag Berlin Heidelberg, New York, pages 123-149, 2004.

Alves A., Arkin A., Askary S., Barreto C., Bloch B., Curbera F., Ford M., Goland Y., Guízar A., Kartha N., Liu C. K., Khalaf R., König D., Marin M., Mehta V., Thatte S., van der Rijn D., Yendluri P., Yiu A., Web Services Business Process Execution Language Version 2.0, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, 11 Avril 2007, (Accès le : 28 Janvier 2015).

Arbache S., Revue de différents Business Model Canvas : M.W. Johnson, BCG, A. Osterwalder et Y. Prigneur, http://businessmodelfab.com/2012/01/07/revue-de-differents-business-model-canvas-m-w-johnson-bcg-a-osterwalder-et-y-prigneur/, 7 Janvier 2012, (Accès le : 10 Avril 2015).

Arsanjani A., "Service-Oriented Modeling and Architecture", *IBM developper works*, 9 Novembre 2004.

Armbrust M., Fox A., Griffith R., Joseph A. D., Katz R., Konwinsko A., Lee G., Patterson D., Rabkin A., Stoica I., Zaharia M., "Above the Clouds: A view of Cloud Computing", ECCS Department, University of California, Berkeley, 2009.

Balas G., APIs for Biz Dev 2.0: Which Business Model ?, http://www.3scale.net/2011/10/apis-for-biz-dev-2-0-which-business-model/, 17 Octobre 2011, (Accès le : 29 Mars 2015).

Banerji A., Bartolini C., Beringer D., Chopella V., Govindarajan K., Karp A., Kuno H., Lemon M., Pogossiants G., Sharma S., Williams S., Web Services Conversation Language (WSCL) 1.0, http://www.w3.org/TR/wscl10/,14 Mars 2002. (Accès le : 28 Janvier 2015).

Bellwood T., Capell S., Clement L., Colgrave J., Dovey M. J., Feygin D., Hately A., Kochman R., Macias P., Novotny M., Paolucci M., von Riegen C., Rogers T., Sycara K., Wenzel P., Wu Z., UDDI Version 3.0.2, UDDI Spec Technical Committee Draft, Dated 20041019, http://uddi.org/pubs/uddi_v3.htm, 19 Octobre 2004, (Accès le : 31 Janvier 2015).

Cabrera F., Copeland G., Cox B., Freund T., Klein J., Storey T., Thatte S., Web Services Transaction (WS-Transaction), http://www.ibm.com/developerworks/library/ws-transpec/, 9 Août 2002, (Accès le : 6 Février 2015).

Castillo P. A., Bernier J. L., Arenas M. G., Merelo J. J., Garcia-Sanchez P., "SOAP vs REST: Comparing a master-slave GA implementation ", *First International Workshop of Distributed Evolutionary computation in Informal Environments*, 25 Mai 2011.

CCSK Guide, Securing an API Strategy, 15 Août 2012, http://ccskguide.org/securing-an-api-strategy/, (Accès le : 11 Août 2015).

Chesbrough H., Rosenbloom R. S., "The role of the business model in capturing value from innovation: evidence from Xerox Corporation's technology spin-off companies", *Industrial and corporate change*, Volume 11, Numéro 3, pages 529-555, 2002.

Chou W., Guo W., Liu F., Zhao Z. Q., "SaaS integration for software cloud", *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 402-409, 10 Juillet 2010.

Christensen E., Curbera F., Meredith G., Weerawarana S., Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl, 25 Mars 2001, (Accès le : 26 Janvier 2015).

Curbera F., N. W. A., and Weerawarana S., "Web services: Why and how.", *OOPSLA 2001 Workshop on Object-Oriented Web Services*, pages 249-259, 9 Août 2001.

Curbera F., Khalaf R., Muhki N., Tai S., Weerawarana S., "The next step in Web Services", *Communication of the ACM – Service-oriented computing*, Volume 46, pages 29-34, Octobre 2003.

Dijkman R. M., Dumas M., Ouyang C., "Semantics and Analysis of Business Process Models in BPMN", *information and Software Technology*, Volume 50, Numéro 12, pages 1281-1294, 29 Février 2008.

Dillon T., Wu C., Chang E., "Cloud computing: issues and challenges", *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 27-33, 23 Avril 2010.

Duvander A., API Business Models, http://www.programmableweb.com/news/api-business-models-then-and-now/2011/05/25, 25 Mai 2011, (Accès le : 28 Mars 2015).

Feingold M., Jeyaraman R., Web Services Coordination (WS-Coordination) Version 1.2, http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec.html, 2 Février 2009, (Accès le : 6 Février 2015).

Gat I., Succi G., "A Survey of the API Economy", *Agile product & project management executive update*, Volume 14, Numéro 6, 2013.

Gudgin M., Hadley M., Mendelsohn N., Moreau J.-J., Nielsen H. F., Karmarkar A., Lafon Y., SOAP Version 1.2 Part 1: Messaging Framework (Second Edition),

http://www.w3.org/TR/soap12-part1/, 27 Avril 2007, (Accès le : 31 Janvier 2015).

Hai H., Sakoda S., “SaaS and integration best practices”, *Fujitsu Science Technology Journal*, Volume 45, Numéro 3, pages 257-264, Juillet 2009.

Hass H., Designing the architecture for Web services, http://www.w3.org/2003/Talks/0521-hh-wsa/slide5-0.html, 22 Mai 2003, (Accès le : 26 Janvier 2015).

Henning M., “API : Design Matters“, *Communication of the ACM*, Volume 52, Numéro 5, pages 46-56, Mai 2009.

Hourdeau F., SaaS, PaaS, IaaS : termes souvent utilisés dans le cadre du « Cloud », http://www.aiservice.fr/aas-paas-iaas-termes-souvent-utilises-dans-le-cadre-du-cloud, 27 Janvier 2013, (Accès le 21 Juillet 2015).

Ide N., Pustejovsky J., "What does interoperability mean, anyway? Toward an operational definition of interoperability for language technology”, *Proceedings of the Second International Conference on Global Interoperability for Language Resources*, 2010.

Inaganti S., Behara G. K., “Service Identification: BPM and SOA Handshake”, *BPTrends*, Volume 3, pages 1-12, Mars 2007.

Investiopedia, http://www.investopedia.com/terms/c/cpm.asp, (Accès le : 28 Mars 2015).

Kana, *Don’t Rip and Replace— Wrap and Renew!*, Sunnyvale, 2013.

Kanmani K. V., Smitha P. S., "Survey on Restful Web Services Using Open Authorization (Oauth)", *IOSR Journal of Computer Engineering*, Volume 15, Numéro 4, pages 53-56, Décembre 2013.

Kim W., Lee J. H., Hong C., Han C., Lee H., Jang B., “An innovative method for data and software integration in SaaS”, *Computers & Mathematics with Applications*, Volume 64, pages 1252-1258, 12 Avril 2012.

Krafzig, D., Banke K., Slama D., *Enterprise SOA: service-oriented architecture best practices*. Prentice Hall Professional, Maryland, 11 Novembre 2004.

Laplante P.A., Zhang J., Voas J., “What's in a Name? Distinguishing between SaaS and SOA. ”, *It Professional*, Volume 10, Numéro 3, pages 46-50, Juin 2008.

Lundquist E., “3 Steps to Success in the API Economy.”, *InformationWeek*, 10 Novembre 2012.

Mäkilä T., Järvi, A., Rönkkö M., Nissilä J., “How to Define Software-as-a-Service–An Empirical Study of Finnish SaaS Providers”, *Software business*, Volume 51, pages 115-124, 23 Juin 2010.

Mell P., Grance T., The NIST Definition of Cloud Computing, http://www.nist.gov/itl/csd/cloud-102511.cfm, 25 Octobre 2011, (Accès le 25 Juillet 2015).

Minnaert G., Sawyer A., Thayer A., Internet-based application program interface (API) documentation interface, https://www.google.com/patents/US6405216, 11 Juin 2002, (Accès le : 8 Février 2015).

MuleSoft, “The 7 habits of highly effective API and service management”, e-book, 2015.

Musser J., API Business Models, API Strategy Conference, New York USA, http://fr.slideshare.net/jmusser/j-musser-apibizmodels2013, 22 Février 2013, (Accès le : 11 Août 2015).

Nadalin A., Kaler C., Hallam-Baker P., Monzillo R., Web Services Security: SOAP Message Security 1.0 (WS-Security 2004), https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss, Mars 2004, (Accès le : 31 Janvier 2015).

Nassif A. B., Capretz M. A.M., “Moving from SaaS Applications towards SOA Services.”, *2010 IEEE 6th Wolrd Congress on* Services, pages 187-188, 10 Juillet 2010.

Open Source Initiative, The Open Source Definition, http://opensource.org/osd, (Accès le : 27 Avril 2015).

Osterwalder A., Pigneur Y., *Business Model nouvelle génération : Un guide pour visionnaires, révolutionnaires et challengers*, Pearson, Paris, 1 Septembre 2011.

Raivio Y., Luukkainen S., Seppäla S., “Towards Open Telco-Business models of API management providers“, *System Sciences (HICSS), 2011 44th Hawaii International Conference on,* pages 1-11, 7 Janvier 2011.

Rouse M., “Stove-piped development definition”, http://searchsoa.techtarget.com/definition/stove-piped-development, (Accès le: 23 février 2015).

Rozlog M., Rest and SOAP : When should I use each (or both) ?, http://www.infoq.com/articles/rest-soap-when-to-use-each ,1 Avril 2010, (Accès le :)

See Technologies, Api Management & development, http://seetechnologies.co/api-management-development.php, (Accès le : Mars et Août 2015).

Sneed H. M., “Integrating legacy software into a service oriented architecture“, *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 11-14, 24 Mars 2006.

Spies B., “Web services, Part 1: SOAP vs. REST“, *Retrieved March*, Volume 12, 2 Mai 2008.

Stylos J., Myers B., “Mapping the space of API design decisions”, *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on*, pages 50-60, 27 Septembre 2007.

Tsai W. T., Sun X., Balasooriya J., “Service-Oriented Cloud Computing Architecture”, *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 684–689, 14 Avril 2010.

Turner M., Budgen D., Brereton P., “Turning software into a service“, *Computer.* , Volume 36, Numéro 10, pages 38-44, 2003.

Vordel, API management platform chart, http://cdn.ttgtmedia.com/rms/onlineImages/sSOA_API_management_0613.jpg, (Accès le : 13 Août 2015).

Vordel, Vordel Survey Finds Half of Enterprises Are Adopting APIs to Build Out New Business Channels, http://www.businesswire.com/news/home/20120522005823/en/Vordel-Survey-Finds-Enterprises-Adopting-APIs-Build#.VRFyrynamEs, 22 Mai 2012, (Accès le : 24 Mars 2015).

Zachman J. A., “A Framework for Information Systems Architecture,” *IBM Systems Journal*, Volume 26, Numéro 3, pages 276–292, 1999.

Zhang L.-J., Zhang J., Cai H., *Services computing*, Springer and Tsinghua University Press, Beijing, Juin 2007.

Zott C., Amit R., Massa R., “The business model: recent developments and future research”, *Journal of management*, Volume 37, Numéro 4, pages 1019-1042, 2011.